Abstract

Building resilient intrusion-tolerant distributed systems is a somewhat complex task. Recently, we have increased this complexity, by presenting a new dimension over which distributed systems resilience may be evaluated — exhaustion-safety. Exhaustion-safety means safety against resource exhaustion, and its concrete semantics in a given system depends on the type of resource being considered. We focus on replicas and on guaranteeing that the typical assumption on the maximum number of replicas failures is never violated. An interesting finding of our work is that it is impossible to build a replica-exhaustion-safe distributed intrusion-tolerant system under the asynchronous model. This result motivated our research on finding the right model and architecture to guarantee exhaustion-safety. The main outcome of this research was proactive resilience — a new paradigm and design methodology to build replica-exhaustion-safe intrusion-tolerant systems under the asynchronous model.

This result motivated our research on finding the right model and architecture to guarantee exhaustion-safety. The main outcome of this research was proactive resilience — a new paradigm and design methodology to build replica-exhaustion-safe intrusion-tolerant distributed systems. Proactive resilience is based on architectural hybridization: the system is asynchronous in its most part and it resorts to a synchronous subsystem to periodically recover the replicas and remove the effects of faults/attacks.

We envisage that proactive resilience can be applied in many different scenarios, namely to secret sharing, and to state machine replication. In the latter context, we present in this paper a novel result that a minimum of \(3f + 2k + 1\) replicas are required for tolerating \(f\) Byzantine faults and maintaining availability, \(k\) being the maximum number of replicas that can be recovered simultaneously through proactive resilience. Different recovery strategies are analyzed in the light of this new result.

1 Achieving Exhaustion-Safety through Proactive Resilience

A distributed system built under the asynchronous model makes no timing assumptions about the operating environment: local processing and message deliveries may suffer arbitrary delays, and local clocks may present unbounded drift rates [13, 7]. Thus, in a (purely) asynchronous system one cannot guarantee that something will happen before a certain time.

Consider now that we want to build a dependable intrusion-tolerant distributed system, i.e., a distributed system able to tolerate arbitrary faults, including malicious ones. Can we build such a system under the asynchronous model?

This question was partially answered, twenty years ago, by Fischer, Lynch and Paterson [8], which proved that there is no deterministic protocol that solves the consensus problem in an asynchronous distributed system prone to even a single crash failure. This impossibility result (commonly known as FLP) has been extremely important, given that consensus lies at the heart of many practical problems, including membership, ordering of messages, atomic commitment, leader election, and atomic broadcast. In this way, FLP showed that the very attractive asynchronous model of computation is not sufficiently powerful to build certain types of fault-tolerant distributed protocols and applications.

What are then the minimum synchrony requirements to build a dependable intrusion-tolerant distributed system?

If the system needs consensus (or equivalent primitives), then Chandra and Toueg [6] showed that consensus can be solved in asynchronous systems augmented with failure detectors (FDs). The main idea is that FDs operate under a more synchronous environment and can therefore offer a service (the failure detection service) with sufficient properties to allow consensus to be solved.

But what can one say about intrusion-tolerant asynchronous systems that do not need consensus? Obviously, they are not affected by the FLP result, but are they dependable?

Independently of the necessity of consensus-like primi-
tives, we have recently shown that relying on the assumption of a maximum number of $f$ faulty nodes under the asynchronous model can be dangerous. Given that an asynchronous system may have a potentially long execution time, there is no way of guaranteeing exhaustion-safety, i.e., that no more than $f$ faults will occur, specially in malicious environments [18]. Therefore, one can rephrase the above statement and say that the asynchronous model of computation is not sufficiently powerful to build any type of (exhaustion-safe) fault-tolerant distributed protocols and applications.

To achieve exhaustion-safety, the goal is to guarantee that the assumed number of faults is never violated. In this context, proactive recovery seems to be a very interesting approach [16]. The aim of proactive recovery is conceptually simple – components are periodically rejuvenated to remove the effects of malicious attacks/faults. If the rejuvenation is performed frequently often, then an adversary is unable to corrupt enough resources to break the system. Proactive recovery has been suggested in several contexts. For instance, it can be used to refresh cryptographic keys in order to prevent the disclosure of too many secrets [12, 11, 10, 24, 4, 23, 15]. It may also be utilized to restore the system code from a secure source to eliminate potential transformations carried out by an adversary [16, 5]. Moreover, it may encompass the substitution of software components to remove vulnerabilities existent in previous versions (e.g., software bugs that could crash the system or errors exploitable by outside attackers). Vulnerability removal can also be done through address space randomization [3, 2, 9, 1, 22], which could be used to periodically randomize the memory location of all code and data objects.

Therefore, proactive recovery has the potential to support the construction of exhaustion-safe intrusion-tolerant distributed systems. However, in order to achieve this, proactive recovery needs to be architected under a model sufficiently strong that allows regular rejuvenation of the system. In fact, proactive recovery protocols (like FDs) typically require stronger environment assumptions (e.g., synchrony, security) than the rest of the system (i.e., the part that is proactively recovered). This is hard or at all impossible to achieve in homogeneous systems’ models [21].

Proactive resilience is a new and more resilient approach to proactive recovery based on architectural hybridization and wormholes [20]. We argue that the proactive recovery subsystem should be constructed in order to assure a synchronous and secure behavior, whereas the rest of the system may even be asynchronous. The Proactive Resilience Model (PRM) proposes to model the proactive recovery subsystem as an abstract component – the Proactive Recovery Wormhole (PRW). The PRW may have many instantiations depending on the application proactive recovery needs.

The architecture of a system with a PRW is suggested in Figure 1. An architecture with a PRW has a local module in every host, called the local PRW. These modules may be interconnected by a synchronous and secure control network. This set up of local PRWs optionally interconnected by the control network is what is collectively called the PRW. The PRW is used to execute proactive recovery procedures of protocols/applications running between participants in the hosts concerned, on any usual distributed system architecture (e.g., the Internet).

Conceptually, a local PRW should be considered to be a module inside a host and separated from the OS. In practice, this conceptual separation between the local PRW and the OS can be achieved in either of two ways: (1) the local PRW can be implemented in a separate, tamper-proof hardware module (e.g., PC appliance board) and so the separation is physical; (2) the local PRW can be implemented on the native hardware, with a virtual separation and shielding between the local PRW and the OS processes implemented in software.

The local PRWs are assumed to be fail-silent (they fail by crashing), and whenever a local PRW fails, the entire node fails permanently. Moreover, the following properties are preserved by construction:

P1 There exists a known upper bound on the processing delays of every local PRW.

P2 There exists a known upper bound on the clock drift rate of every local PRW.

Additionally, if the local PRWs are interconnected by a synchronous and secure control network, the following property is guaranteed:

P3 There exists a known upper bound on message delivery delays.

Figure 1. The architecture of a system with a PRW.
The control network is not always necessary. For instance, if a proactive recovery procedure only requires local information, then the control network is expendable.

2 Application Scenarios

2.1 Secret Sharing

Secret sharing schemes protect the secrecy and integrity of secrets by distributing them over different locations. A secret sharing scheme transforms a secret $s$ into $n$ shares $s_1, s_2, ..., s_n$ which are distributed to $n$ share-holders. In this way, the adversary has to attack multiple share-holders in order to learn or to destroy the secret. For instance, in a $(k+1, n)$-threshold scheme, an adversary needs to compromise more than $k$ share-holders to learn the secret, and corrupt at least $n-k$ shares in order to destroy the same secret.

Various secret sharing schemes have been developed to satisfy different requirements. We use Shamir's scheme [17] to implement a $(k+1, n)$-threshold scheme: given an integer valued secret $s$, pick a prime $q$ which is bigger than both $s$ and $n$. Randomly choose $a_1, a_2, ..., a_k$ from $[0, q]$ and set polynomial $f(x) = (s + a_1 x + a_2 x^2 + ... + a_k x^k) \mod q$. For $i = 1, 2, ..., n$, set the share $s_i = f(i)$. The reconstruction of the secret can be done by having $k+1$ participants providing their shares and using polynomial interpolation to compute $s$. Moreover, given $k$ or fewer shares, it is impossible to reconstruct $s$.

A special case where $k = 1$ (that is, two shares are required for reconstructing the secret), is given in Figure 2. The polynomial is a line and the secret is the point where the line intersects with the y-axis. Namely, this point is the point $(0, f(0)) = (0, s)$. Each share is a point on the line. Any two (i.e., $k+1$) points determine the line and hence the secret. With just a single point, the line can be any line that passes the point, and hence the secret can be any point on the y-axis.

In many applications, a secret $s$ may be required to be held in a secret-sharing manner by $n$ share-holders for a long time. If at most $k$ share-holders are corrupted throughout the entire lifetime of the secret, any $(k+1, n)$-threshold scheme can be used. In certain environments, however, gradual break-ins into a subset of locations over a long period of time may be feasible for the adversary. If more than $k$ share-holders are corrupted, $s$ may be stolen. An obvious defense is to periodically refresh $s$, but this is not possible when $s$ corresponds to inherently long-lived information (e.g., cryptographic root keys, legal documents).

Thus, what is actually required to protect the secrecy of the information is to be able to periodically renew the shares without changing the secret. Proactive secret sharing (PSS) was introduced in [12] in this context. In PSS, the lifetime of a secret is divided into multiple periods and shares are renewed periodically. In this way, corrupted shares will not accumulate over the entire lifetime of the secret since they are checked and corrected at the end of the period during which they have occurred. A $(k+1, n)$ proactive threshold scheme guarantees that the secret is not disclosed and can be recovered as long as at most $k$ share-holders are corrupted during each period, while every share-holder may be corrupted multiple times over several periods.

Recently, we showed that it is possible to build a highly resilient distributed intrusion-tolerant secret sharing system through the use of a specific instantiation of the PRW [20]. Initial experimental results show that this system is able to resist any number of intrusions produced at the maximum rate of one intrusion per second, over the lifetime of the system.

2.2 State Machine Replication

In a previous work [19], we explain how proactive resilience can be used to build an exhaustion-safe state machine replication system. We also give the intuition that the redundancy quorum to tolerate Byzantine faults should have in account the number of simultaneous recoveries triggered through proactive resilience. More recently, we have found precisely how much extra redundancy is needed. The next section presents a brief explanation of our result, and its consequences in terms of the possible recovery strategies.

2.2.1 Why $n \geq 3f + 2k + 1$?

Consider that you have a replicated state machine replication system with $n$ replicas, able to tolerate a maximum of $f$ Byzantine faults, and where rejuvenations occur in groups of at most $k$ replicas. At any time, the minimum number of replicas assuredly available is $n - f - k$. So, in any operation, either intra-replicas (e.g., a consensus execution),
or originated from an external participant (e.g., a client request), a group with \( n - f - k \) replicas will be used to execute the operation. Given that some of these operations may affect the state of the replicated system, one also needs to guarantee that any two groups of \( n - f - k \) replicas intersect in at least \( f + 1 \) replicas (i.e., one correct replica). Therefore, one needs to guarantee that \( 2(n - f - k) - n \geq f + 1 \), which can only be satisfied if \( n \geq 3f + 2k + 1 \).

The reasoning above can be seen in practice by analyzing the Byzantine dissemination quorum system construction [14], which applies to replicated services storing self-verifying data, i.e., data that only clients can create and to which clients can detect any attempted modification by a faulty server (e.g., public key distribution system). In a dissemination quorum system, the following properties are satisfied:

**Intersection** any two quorums have at least one correct replica in common;

**Availability** there is always a quorum available with no faulty replicas.

If one designates \(|Q|\) as the quorum size, then the above properties originate the following conditions:

**Intersection** \( 2|Q| - n \geq f + 1 \);

**Availability** \(|Q| \leq n - f - k \).

From these conditions, it results that one needs \( n \geq 3f + 2k + 1 \) in order to have a dissemination quorum system in an environment where at most \( f \) replicas may behave arbitrarily, and at most \( k \) replicas may recover simultaneously (and thus become unavailable during certain periods of time). In the special case when \( n = 3f + 2k + 1 \), it follows that \(|Q| = 2f + k + 1|\).

### 2.2.2 Recovery Strategies

The challenge of the system architect is to choose the concrete values of \( k, f, \) and \( n \), which are more appropriate according to the specific requirements of the service offered by the replicated state machine. In the light of the result presented in the previous section, an important design parameter that should be considered is the value of \( k \). \( k \) defines the number of nodes that may recover simultaneously, and consequently the number of distinct \( \lceil n/k \rceil \) replica groups that recover in sequence. Intuitively, choosing a higher \( k \), reduces the number of rejuvenation groups, but increases the total number of nodes \( n \) (required to guarantee availability during recoveries). On the other hand, a smaller \( k \) means more recoveries in sequence, but a lower \( n \). Typically, each rejuvenation group will reboot and execute state transfer. Both these operations may take a considerable amount of
time, and therefore the system architect should try to minimize the number of rejuvenation groups. This being said, let us analyze how the value of \( \lceil n/k \rceil \) evolves in different scenarios.

First of all, a minimum of \( 3 \) rejuvenation groups will exist in every configuration with \( k \geq 1 \), i.e., either the replicas do not rejuvenate at all \((k = 0)\), or if they rejuvenate, at least a sequence of \( 3 \) rejuvenations will occur. This follow from the fact that \( n \geq 3f + 2k + 1 \), and thus \( n/k \geq 2 \).

Secondly, the number of rejuvenation groups is upper-bounded by \( n \). In the worst case, \( k = 1 \), and therefore \( \lceil n/k \rceil = n \). So, we see that the number of rejuvenation groups may vary between \( 3 \) and \( n \), depending on the value of \( k \). In fact, it is easy to see that in order to have a certain number \( l = \lceil n/k \rceil \) of rejuvenation groups, one has to choose \( k \geq \frac{3f + 1}{l - 1} \). A special case of this condition is that the minimum number of rejuvenations \((l = 3)\), is obtained by setting \( k \geq 3f + 1 \). For instance, if \( f = 1 \), and one wants \( l = 3 \) rejuvenation groups, then \( k \) should be greater than \( 4 \).

Figure 3 depicts the evolution of the number of rejuvenation groups resulting from combinations of \( k \) and \( f \) values. As expected, the number \( l \) of rejuvenation groups is lower for higher values of \( k \), and \( l = 3 \) for every \( k \geq 3f + 1 \).

Figure 4 shows the cost, in terms of the minimum number of replicas required, to guarantee \( n \geq 3f + 2k + 1 \). We see that \( n \) linearly increases with \( k \), at a rate that is independent of the specific value of \( f \).

These last two figures can help the system architect to calculate the cost of choosing a specific \( f \) and \( k \), in terms of the total number of replicas and the number of rejuvenation groups. A different perspective is presented in Figure 5. In this figure, one can analyze what is the minimum number of rejuvenation groups allowed by a given \( n \) and \( f \). It can help the system architect to decide if it pays to tolerate a higher \( k \) in order to decrease the number of rejuvenation groups. For instance, with \( n = 13 \) and \( f = 3 \), a minimum of 13 rejuvenation groups is required. (because


$k = 1$), but with a single one more replica ($n = 14$), only 7 rejuvenations in sequence are needed, because $k$ can now be set to 2. If one continues to follow the line representing $f = 3$, the conclusion is that adding still one more replica ($n = 15$) would rather worsen recoveries than improving it, given that both the number of replicas and rejuvenation groups increase. The general conclusion is that, if one starts from a configuration with $k = 1$, the higher gain in terms of the number of rejuvenation groups is achieved by adding the necessary number of replicas such that $k$ can be set to 2. By doing this, the number of rejuvenation groups is reduced from $n$ to $\lceil n/2 \rceil$. In practice, if the system is deployed with $n = 3f + 2k + 1 = 3f + 3$ replicas because $k = 1$, then one should add two more replicas to allow $k = 2$. These two extra replicas decrease substantially the minimum number of rejuvenation groups. This decrease is proportional to the value of $f$. For instance, with $f = 1$, the decrease is from 6 to 4 rejuvenation groups, but with $f = 5$, the decrease is from 18 to 10 rejuvenations in sequence.

3 Future Work

We are in the process of implementing an experimental prototype of a proactive resilient state machine replication system. The goal is to compare, in practice, the resilience of such a system with the resilience of similar ones built using different approaches.

Another objective we are pursuing is to research ways of using proactive resilience to mitigate Denial-of-Service attacks.

References


