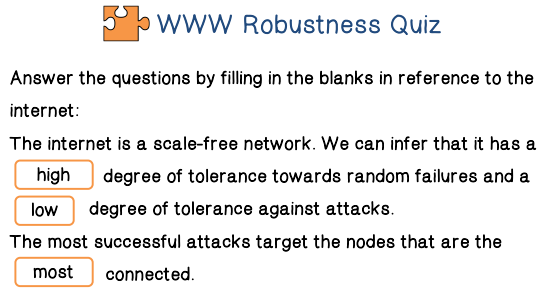
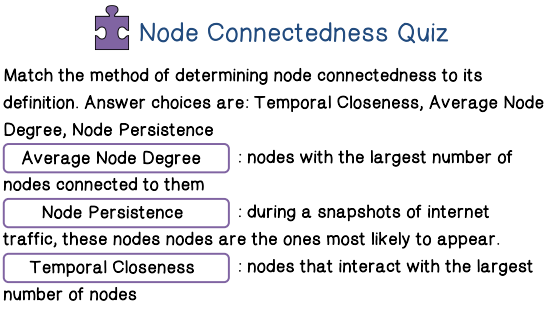
We have networks with valuable data on them. These networks have been attacked or will be attacked. We can do everything humanly possible to stop attacks, but we also need to build systems that can withstand attacks. In this lesson, we learned about attack tolerant systems.



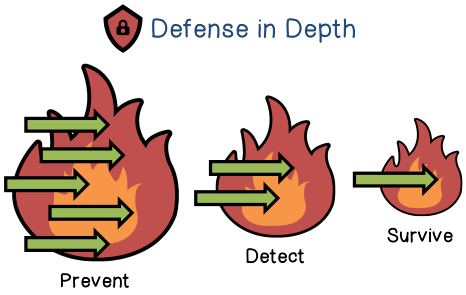
As we all know, the Internet is vulnerable to attacks. Since we are going to discuss attack tolerant systems, let's first make sure we understand how the structure of the Internet is both a blessing and a curse. In this quiz, answer the questions in reference to the Internet.

The internet is a scale-free network. We can infer that it has a high degree of tolerance towards random failure and a low degree of tolerance against attacks. The most successful attacks target the nodes that are the most connected. When a highly connected node is attacked, the internet begins to fracture and splinter into unconnected networks. It is this characteristic of the internet that makes attacks so successful.

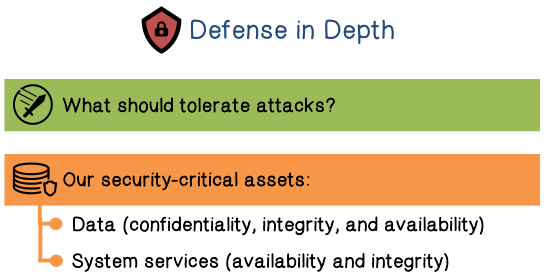


Determining which nodes are the most connected is an interesting problem. There are basically three different methods of determining node connectivity. In this quiz, match the method of determining no connectivity to its definition.

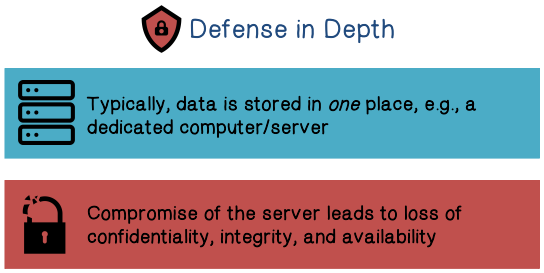
Average node degree, look at nodes with the largest number of nodes connected to them. Node persistence, look at a snapshot of the internet traffic, these nodes are the ones that are most likely to appear. Temporal closeness, look at nodes that interact with the largest number of nodes.



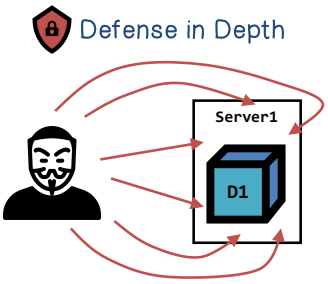
We all know the principle of defense in depth. That is, the first layer of defense is prevention. The second layer is detection, and the third layer is about surviving the attack. The best way to survive an attack is to able to tolerate the attack. That is, an attack will not be able to render our network or system ineffective.



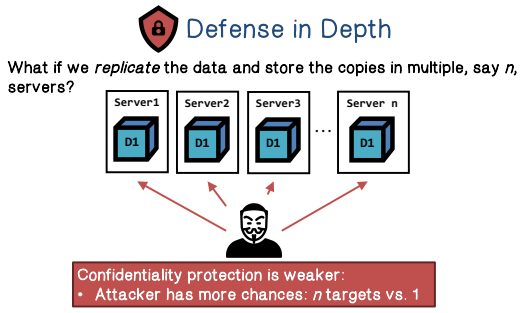
When we say our network and systems should tolerate an attack, we mean that our data and our services should tolerate the attacks. For our data, we want the confidentiality, integrity, and availability to remain intact. For our system services, we want the availability and integrity to remain intact.



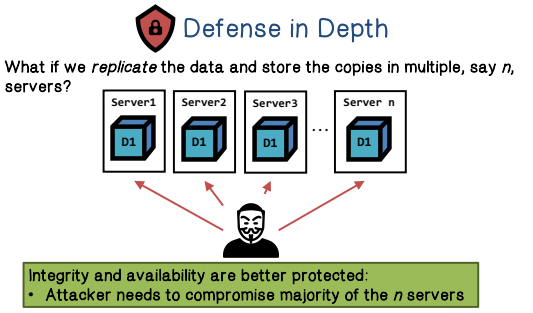
Let's first look at how to make our data tolerate an attack. Typically, data is stored in one place, for example, a dedicated server. So, all the attacker needs to do is to compromise that data storage server.



That is, if the attacker can compromise the data storage server, then the confidentiality, integrity, and availability of data are all compromised.

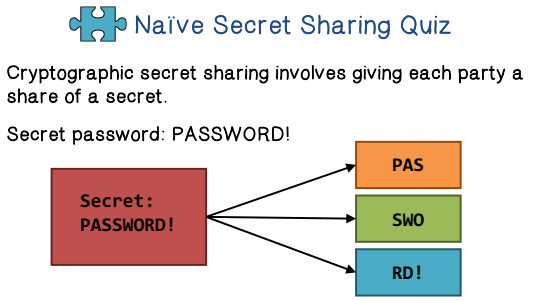


Now, what if we replicate the data and store the copies in multiple, say n different servers, would this scheme improve security? That is, would this scheme make our data more tolerant towards attacks? For confidentiality protection, this is actually a weaker scheme because the attacker can get data from any of these n servers. That is, the attacker now has more opportunities.

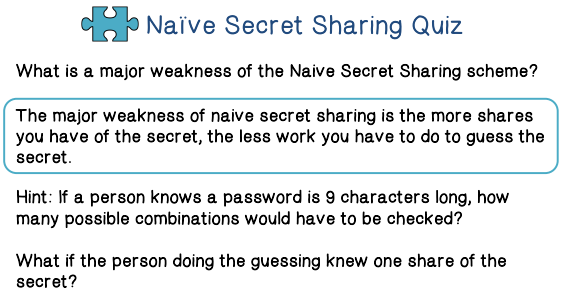


Whereas, if we use a majority voting scheme, then integrity and availability are better protected because now the attacker needs to compromise the majority of these n servers.

So, now the question is, for our data, can we have a better confidentiality protection?

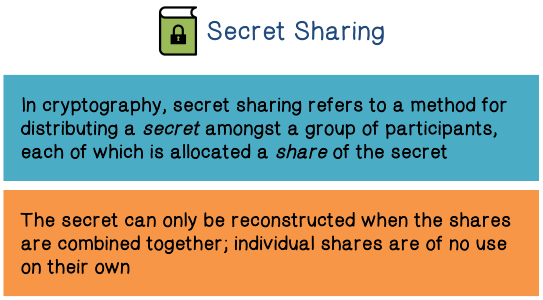


One approach is by secret sharing. Cryptographic secret sharing involves, giving each party a share of a secret, the secret can only be reconstructed if all parties participate and share their portion. For example, if the secret is a password. That is P-A-S-S-W-O-R-D! we can break the secret into shares, and give the shares to three different parties. That is, now each of these three parties has a share of the secret.

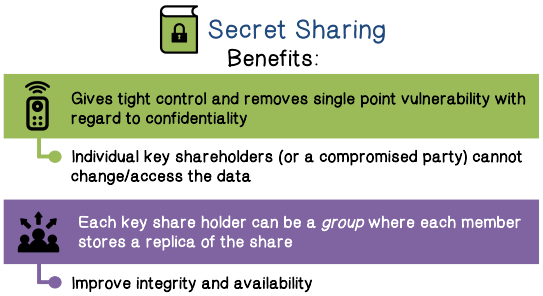


Now, I'm going to ask you what's wrong with this scheme, I'll give you a hint. If the attacker knows the password has nine characters, how many possible combinations will have to be checked? What if the attacker doing the guessing, knew one share of the secret.

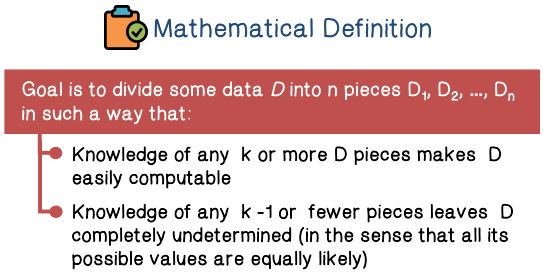
The major weakness of this naive secret sharing scheme is that the more shares you have of the secret, the less work you have to do to guess the secret. In a secure scheme, it should not matter how many shares of a secret a party has. You should take the same amount of guess work as a party with no shares.



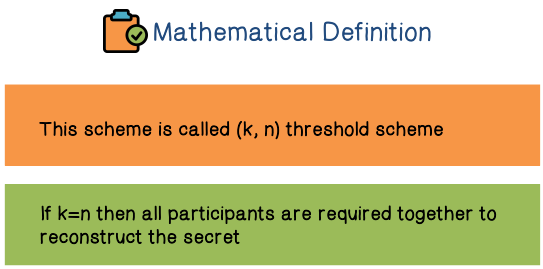
Again, in cryptography, a secret sharing scheme distributes the secret among a group of participants, so that each of the participant has a share of the secret. The share can only be reconstructed when the shares are combined together. Most importantly, individual shares are of no use on their own.



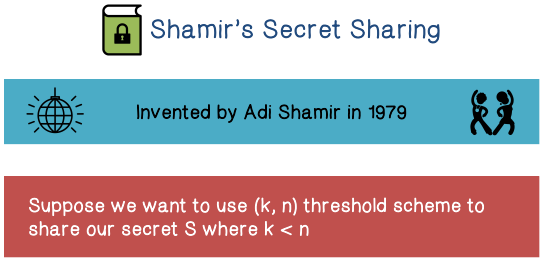
We secure sharing. We can give a tighter control and remove any single point of vulnerability with regard to confidentiality. This is because even if an attacker has compromised an individual key shareholder he still cannot change or access to data. And of course, we can also improve integrity and availability by replicating each share among the group.



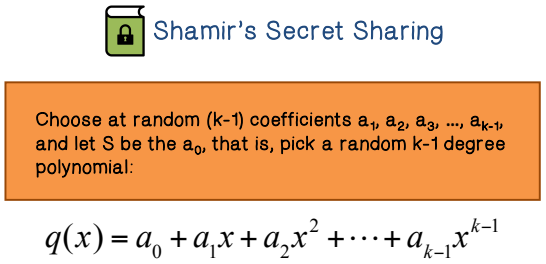
Now, let's discuss the math behind secure sharing. The goal of secure sharing is that for some secret data say D, we divide the data into n pieces, D1, D2, through Dn. In such a way that, if you know k or more pieces of D, then you can compute D. On the other hand, if you know only k minus one of fewer pieces of D, then you can not determine D. The best you can do is a random guess.



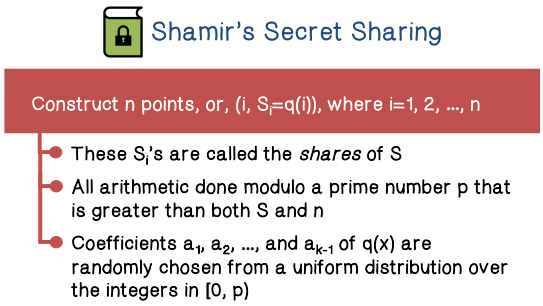
This is the so-called k, n threshold scheme. If k equal to n, that means all participants are required together to reconstruct the secret.



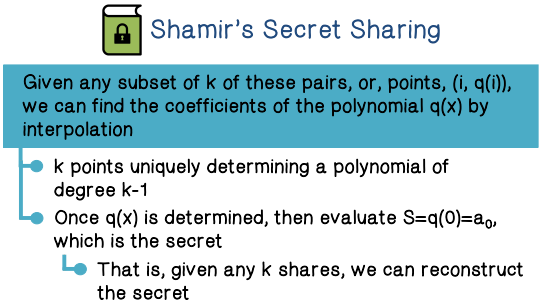
Now let's discuss a threshold secret sharing scheme by Shamir. Shamir is the S in RSA and in 1979, he invented a (k,n) threshold scheme.



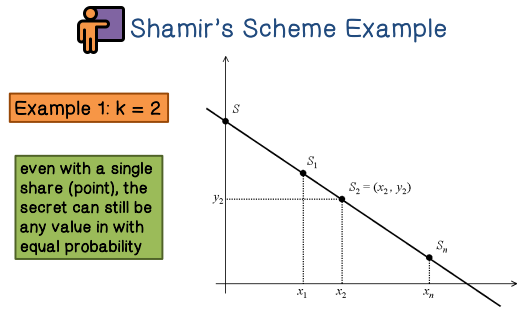
Here's how Shamir's secret sharing scheme works. We choose a random k-1 coefficients; a1, a2, a3 through a_k1. We let the secret S be a0. In other words, we pick a random k-1 degree polynomial and we represent it as this. Again, a0 is a secret S, and a1, a2 through a_k-1 are randomly chosen.



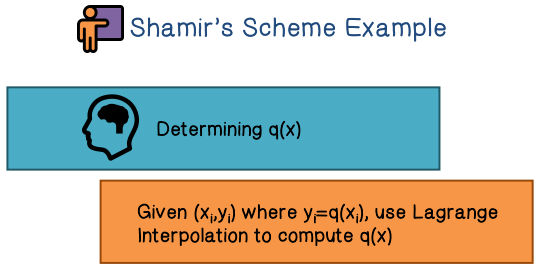
Then we construct n points on this polynomial. That is, for i=1, 2 all the way through n, we compute q of i. Then we construct n points of the polynomial q(x). That is, for i=1, 2, all the way through n, we compute Si equal to q of i. For example, if i=1, we set x=1 for q(x). This Si's are essentially the shares of S. All arithmetic operations are done modulo a prime number p, and p is greater than S and n. Again, the coefficients a1, a2, through a_k-1 are randomly chosen from a uniform distribution over the integers in the range of 0 and p.



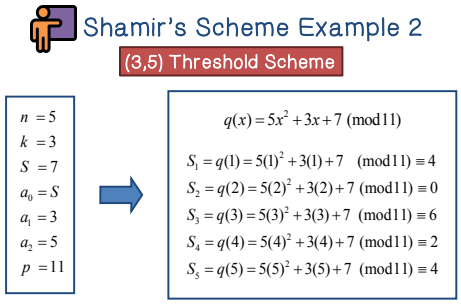
Given any subset of k of these pairs or points, we can find the coefficients of the polynomial q(x) by interpolation. Of course, once we find the coefficients, we have determined the polynomial q(x). This is guaranteed by the mathematical property that any k points can uniquely determine a polynomial of degree k-1. Once q(x) is determined, then we set x equal to 0 and compute q(0), that will give us a0, which is the original secret S. In other words, with this scheme given any K shares, we can reconstruct the secret.



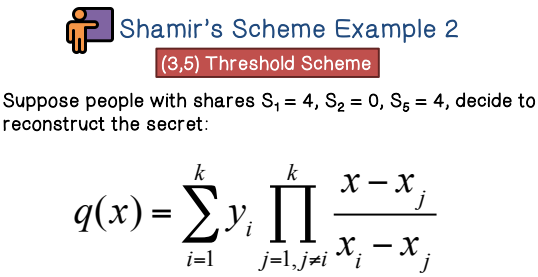
Here's an example. Suppose k equal to two. That is, the polynomial is a linear function, a line. Suppose all you know is one point, say S1. You cannot determine the polynomial because there are infinite number of lines that go through S1. It is clear that you need two points to determine a line. For example, if you know both S1 and S2, then you can reconstruct the line and then you can compute S, which is the original secret. Notice that, this example also shows that obviously, any two points on a line can determine the same line, which means any two of the entries can help you reconstruct the original secret S.



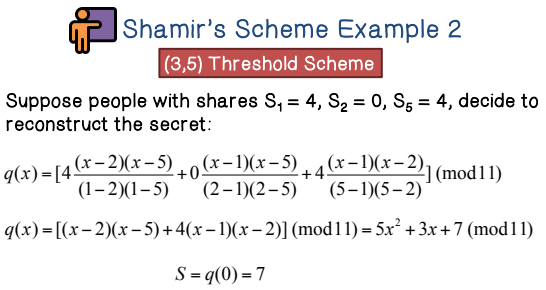
In general, when we need to determine q(X), we should use the Lagrange interpolation algorithm.



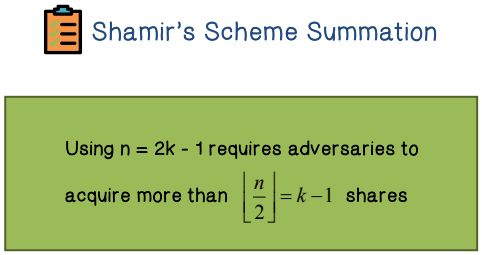
Let's use an example to illustrate. Suppose k is three and n is five. Further, the ultra secret is seven and a0 is seven, a1 is three and a2 is five and the modular prime number p is 11. That is; q(x) is 5x squared plus 3x plus seven modular 11. Since n is five, we compute five shares of five points by setting x to be one, two, three, four, and five, and these are the shares.



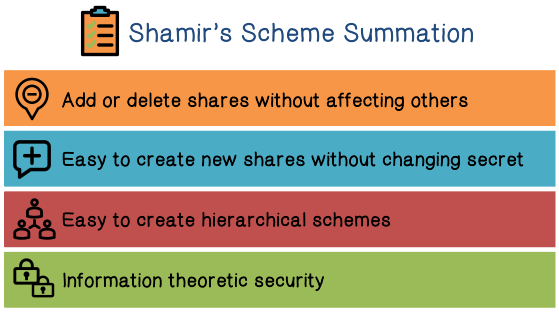
Since k is three with three shares, we should be able to reconstruct the secret. Say we have three shares S1, S2, and S5. Then we can use this formula due to Lagrange to compute q(x).



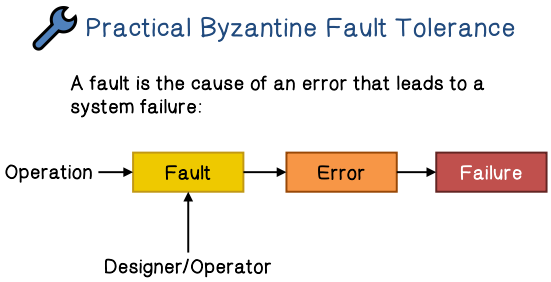
So, we can plug in these points to the formula that is setting the x, i or j's to be one, two and five and setting the y, i's to be four, zero, and four. Then we have q(x) represented as this formula. With simple arithmetic, you can verify that q(x) is five x squared plus 3x plus seven. Then we set x to zero and q of zero is seven which is the original secret and this is how we use Shamir's Secret Sharing Scheme.



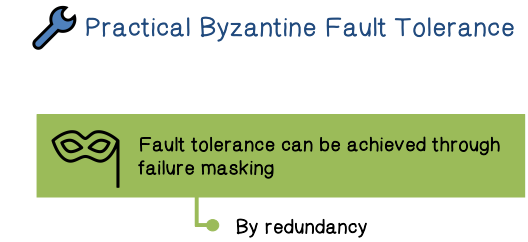
In practice, if we set n equal to 2k minus one, that will require an attacker to compromise more than half of the shares in order to reconstruct the original secret. In general, we typically assume that an attacker cannot compromise the majority of our systems.



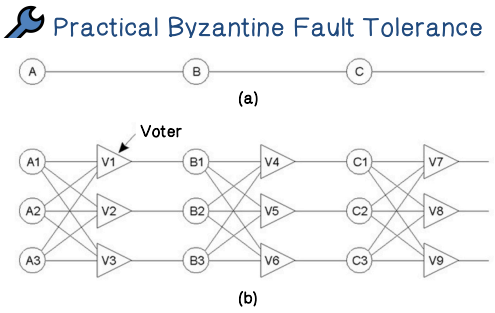
Let's summarize the security properties of Shamir's secret sharing scheme. First of all, shares can be dynamically added or deleted without affecting the other shares. Second, security can be easily enhanced without changing a secret. For example, we can change the polynomial, recompute the shares, and then, give the new shares to the participants. Third, in organizations where hierarchy is important, we can supply each participant different number of shares according to their importance. For example, the CEO can have all the shares so that he alone can reconstruct the secret, whereas three VPs are required together to reconstruct the secret. Fourth, this scheme cannot be broken even when the adversary has unlimited computing power. That is, the attacker has to have k shares. Otherwise, no matter how much power he has, he can only be a random guest.



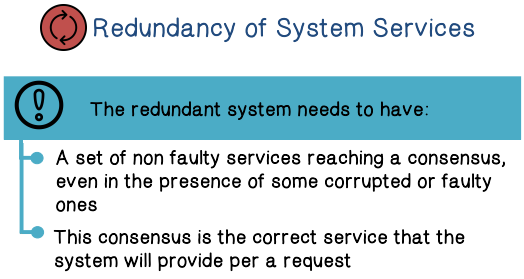
Now let's discuss how do we make system services tolerate attacks. First, we will discuss a related area called fault tolerance. A fault in the system is the cost of an error that leads to a system failure. Again, a system error is because of a fault, and error leads to a failure.



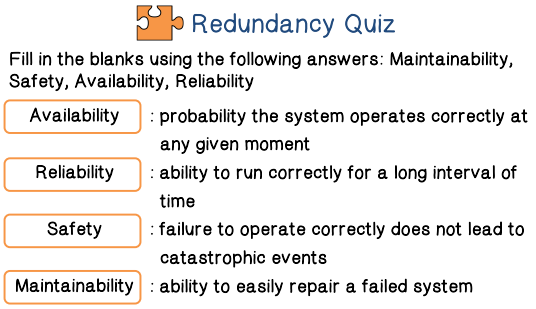
Fault tolerance can be achieved through a failure masking. And failure masking can be achieved through redundancy.



For example, let's look at an example system. Here, a single fault in any of these three components can lead to a failure. An example of redundancy is triple modular redundancy. We can put motors at each stage of the system. And each voter accepts the majority as the correct output from the components. This scheme can mask any single failure of the components.

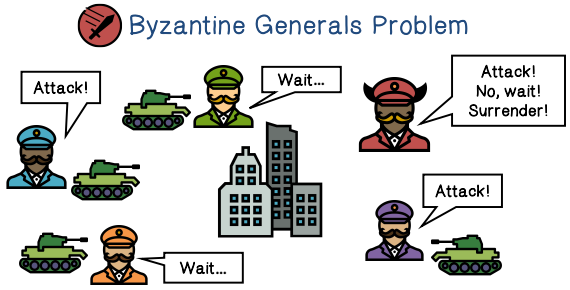


In a previous example of a redundant system, we see that the motor takes the majority as the correct output. This assumes that the non-faulty components can reach a consensus. Therefore, the goal of a redundant system is that a set of non-faulty services can reach a consensus, even in the presence of some corrupted or faulty services. This consensus is then the correct service that a system will provide.

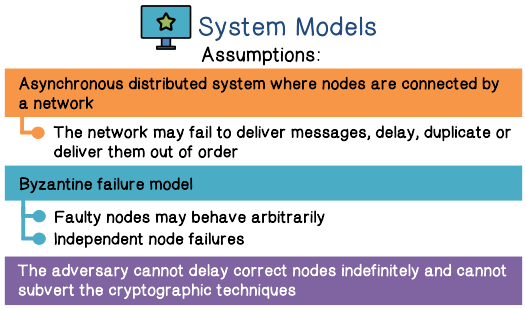


Now, let's do a quiz. Match these terms with their descriptions.

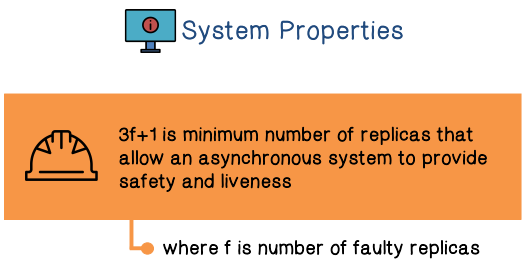
Availability measures how likely a system can operate correctly at any given moment. Reliability is the ability for system to run correctly for longtime. Safety means that, failure to operate correctly does not lead to catastrophic events. Maintainability is the ability to easily repair a failed system.



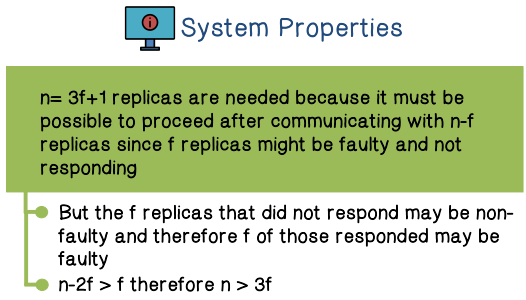
Getting a set of non-faulty services to reach consensus is actually quite challenging. Let's use an example to illustrate. The Byzantine Generals Problem refers to a hypothetical situation in which a group of generals, each commanding a portion of an army plan to attack a city. The generals must decide to attack or wait. Every General must agree on a common decision, anything less will result in a defeat. The problem is compounded by the fact that there may be traitorous generals that do not vote for the best strategy. Of course, there can be disagreement among the generals. For example, some would vote for attack, while others will go for wait. As you can see, it's not easy to reach consensus.



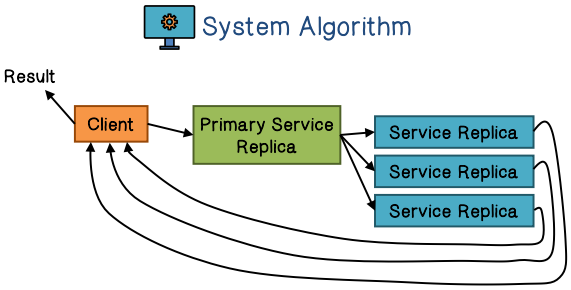
The Byzantine generals problem is an example of asynchronous distributed system. In such systems, we typically assume that the network is not reliable. That is, there's no guarantee for reliable communications. Furthermore, a faulty node may behave arbitrarily. That means, its behavior is not defined. Also, the nodes fail independently. That means, there's no correlation of failures among the nodes. On the other hand, we also assume that the attackers cannot indefinitely block a node from providing service, and the attacker cannot break crypto.



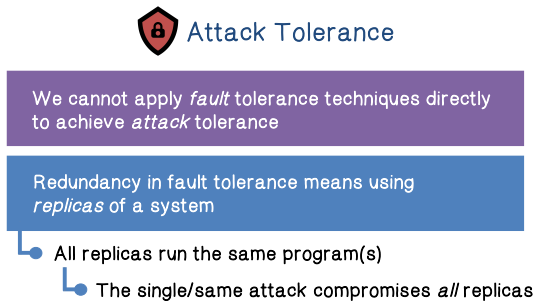
When we say, an asynchronous system achieves fault-tolerance, we typically mean that, it has both safety and liveness. Safety means that, even if the system fails, nothing series happens. Liveliness means that, the kinds of the system can eventually receive replies to their requests. In order for an asynchronous system to provide safety and liveness, it needs to have a minimum 3f plus one replicas. Here, f is the maximum number of faulty replicas.



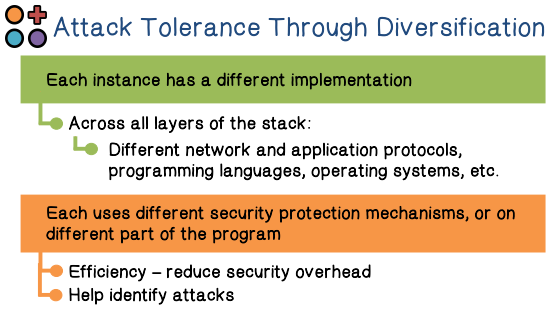
We need 3f plus one replicas, it must be possible to proceed after communicating with n minus one replicas, since f replicas might be faulty and not responding. However, it is possible that the f replicas that did not respond are not faulty, and therefore f of those that respond, might be faulty. Even so, there must be enough responses, that those from non-faulty replicas outnumber those from faulty ones, that is, n minus two f is greater than f. Therefore, n has to be greater than 3f.



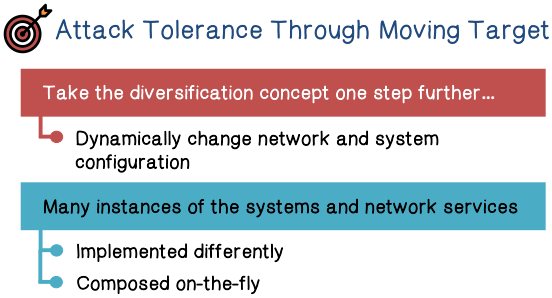
So here is a sketch of how a client can get service from this system. A client sends a request to invoke a service operation to the primary service replica. The primary multicasts the request to the backups, meaning the group of replicas. The replicas execute the request and send a reply to the client. The client waits for f plus one replies from different replicas with the same result, and that is the result of the operation.



So far we have discussed fault tolerance for system services, but what about attack tolerance for system services? Can we apply fault tolerance techniques to achieve attack tolerance? The answer is no. Because in fault tolerance, we use redundancy which means we use replicas of a system and all replicas runs the same program. That means the same attack can compromise all replicas. Therefore, redundancy is not a solution for attack tolerance.



To achieve attack tolerance, we need to use diversification. That is each instance to use a different implementation and this applies to all layers of the stack. Which can include network and application protocols, programming languages, operating systems and so on. Each instance can use a different security protection mechanism or apply a security mechanism to different part of the program. This can achieve efficiency for example because not all operations are checked all the time. It can also help us identify the attacks based on which protection mechanism works and which fails. Of course, it is very costly to implement and very complex to manage diversification.



We can take the idea of diversification one step further to use what we call moving target techniques to achieve attack tolerance. That is, we can dynamically change our network and system configurations. For example, we can have many instances of the system and network services. Each instance can have a different implementation, and these instances can be composed on-the-fly.

How to Share a Secret

In this paper it is shown how to divide data D into n pieces in such a way that D is easily re-constructible from any k pieces, but even complete knowledge of k - 1 pieces reveals absolutely no information about D. This technique enables the construction of robust key management schemes for cryptographic systems that can function securely and reliably even when misfortunes destroy half the pieces and security breaches expose all but one of the remaining pieces.

In the paper, the problem is generalized to one in which the secret is some data D (e.g., the safe combination) and in which nonmechanical solutions (which manipulate this data) are also allowed. The goal is to divide D into n pieces D, ... D n in such a way that:

(1) knowledge of any k or more D i pieces makes D easily computable;

(2) knowledge of any k- 1 or fewer Di pieces leaves D completely undetermined (in the sense that all its possible values are equally likely).

Such a scheme is called a **(k, n) threshold scheme. [1]**

Efficient threshold schemes can be very helpful in the management of cryptographic keys. In order to protect data, we can encrypt it, but in order to protect the encryption key we need a different method (further encryptions change the problem rather than solve it). The most secure key management scheme keeps the key in a single, well-guarded location (a computer, a human brain, or a safe). This scheme is highly unreliable since a single misfortune (a computer breakdown, sudden death, or sabotage) can make the information inaccessible. An obvious solution is to store multiple copies of the key at different locations, but this increases the danger of security breaches (computer penetration~ betrayal, or human errors). By using a (k, n) threshold scheme with n = 2k- 1 we get a very robust key management scheme: We can recover the original key even when [n/2J = k- 1 of the n pieces are destroyed, but our opponents cannot reconstruct the key even when security breaches expose [n/21 = k- 1 of the remaining k pieces.

Threshold schemes are ideally suited to applications in which a group of mutually suspicious individuals with conflicting interests must cooperate. Ideally, we would like the cooperation to be based on mutual consent, but the veto power this mechanism gives to each member can paralyze the activities of the group. By properly choosing the k and n parameters we can give any sufficiently large majority the authority to take some action while giving any sufficiently large minority the power to block it. [2]

[1] Refer to "A simple (k,n) Threshold scheme section for details." [2] Refer to Introduction for detailed explanation

Practical Byzantine Fault Tolerance

The paper described a new replication algorithm that is capable of tolerating Byzantine faults. A Byzantine fault is a condition in a computer system, particularly distributed systems, where components may fail and there is imperfect information on whether a component has failed. The authors of the paper believe that Byzantine fault-tolerant algorithms will be increasingly important in the future because malicious attacks and software errors are increasingly becoming common and can cause faulty nodes to exhibit arbitrary behavior.

The paper makes the following contributions:

1. It describes the first state-machine replication protocol that correctly survives Byzantine faults in asynchronous networks.
2. It describes a number of important optimizations that allow the algorithm to perform well so that allow the algorithm to perform well so that it can be used in real systems.
3. It describes the implementation of a Byzantine-fault-tolerant distributed file system.
4. It provides experimental results that quantify the cost of the replication technique.

The paper assumes an asynchronous distributed system where nodes are connected by a network(Section2), this leads to the need for a Byzantine failure model. The algorithm that is discussed in the paper can be used to implement any deterministic replicated service with a state and some operations. The algorithm used can be summarized using the following steps(Section4):

1. A client sends a request to invoke a service operation to the primary.
2. The primary multicasts the requests to the backups.
3. Replicas execute the request and send a reply to the client.
4. The client waits for f + l replies from the different replicas with the same result; this is the result of the operation.

Section 4 explains in detail the normal-case operation, garbage collection, view changes and correctness of the algorithm. The correctness includes aspects such as safety, liveness and non-determinism (Section 4.5). This is followed by discussions of optimization for the algorithm, these optimizations improve the performance of the algorithm during normal-case operation. The optimizations discussed include reducing communication by avoiding sending large replies and reduce the number of message delays for an operation invocation (Section 5.1).