someone might do a large-scale service-denial attack by convincing millions of machines that their owners had tampered with the clock, causing their files to become inaccessible.

Anyway, there are several possible approaches to the provision of secure time.

- You could furnish every computer with a radio clock, but that can be expensive, and radio clocks — even GPS — can be jammed if the opponent is serious.
- There are clock synchronization protocols described in the research literature in which a number of clocks vote in a way that should make clock failures and network delays apparent. Even though these are designed to withstand random (rather than malicious) failure, they can no doubt be hardened by having the messages digitally signed.
- You can abandon absolute time and instead use *Lamport time* in which all you care about is whether event A happened before event B, rather than what date it is [766]. Using challenge-response rather than timestamps in security protocols is an example of this; another is given by timestamping services that continually hash all documents presented to them into a running total that's published, and can thus provide proof that a certain document existed by a certain date [572].

However, in most applications, you are likely to end up using the *network time protocol* (NTP). This has a moderate amount of protection, with clock voting and authentication of time servers. It is dependable enough for many purposes.

6.3 Fault Tolerance and Failure Recovery

Failure recovery is often the most important aspect of security engineering, yet it is one of the most neglected. For many year, most of the research papers on computer security have dealt with confidentiality, and most of the rest with authenticity and integrity; availability has been neglected. Yet the actual expenditures of a typical bank are the other way round. Perhaps a third of all IT costs go on availability and recovery mechanisms, such as hot standby processing sites and multiply redundant networks; a few percent more get invested in integrity mechanisms such as internal audit; and an almost insignificant amount goes on confidentiality mechanisms such as encryption boxes. As you read through this book, you'll see that many other applications, from burglar alarms through electronic warfare to protecting a company from Internet-based service denial attacks, are fundamentally about availability. Fault tolerance and failure recovery are a huge part of the security engineer's job.

Classical fault tolerance is usually based on mechanisms such as logs and locking, and is greatly complicated when it must withstand malicious attacks on these mechanisms. Fault tolerance interacts with security in a number of ways: the failure model, the nature of resilience, the location of redundancy used to provide it, and defense against service denial attacks. I'll use the following definitions: a *fault* may cause an *error*, which is an incorrect state; this may lead to a *failure* which is a deviation from the system's specified behavior. The resilience which we build into a system to tolerate faults and recover from failures will have a number of components, such as fault detection, error recovery and if necessary failure recovery. The meaning of *mean-time-before-failure* (MTBF) and *mean-time-to-repair* (MTTR) should be obvious.

6.3.1 Failure Models

In order to decide what sort of resilience we need, we must know what sort of attacks are expected. Much of this will come from an analysis of threats specific to our system's operating environment, but there are some general issues that bear mentioning.

6.3.1.1 Byzantine Failure

First, the failures with which we are concerned may be normal or *Byzantine*. The Byzantine fault model is inspired by the idea that there are n generals defending Byzantium, t of whom have been bribed by the Turks to cause as much confusion as possible in the command structure. The generals can pass oral messages by courier, and the couriers are trustworthy, so each general can exchange confidential and authentic communications with each other general (we could imagine them encrypting and computing a MAC on each message). What is the maximum number t of traitors which can be tolerated?

The key observation is that if we have only three generals, say Anthony, Basil and Charalampos, and Anthony is the traitor, then he can tell Basil 'let's attack' and Charalampos 'let's retreat'. Basil can now say to Charalampos 'Anthony says let's attack', but this doesn't let Charalampos conclude that Anthony's the traitor. It could just as easily have been Basil; Anthony could have said 'let's retreat' to both of them, but Basil lied when he said 'Anthony says let's attack'.

This beautiful insight is due to Leslie Lamport, Robert Shostak and Marshall Pease, who proved that the problem has a solution if and only if $n \geq 3t + 1$ [767]. Of course, if the generals are able to sign their messages, then no general dare say different things to two different colleagues. This illustrates the power of digital signatures in particular and of end-to-end security mechanisms in general. Relying on third parties to introduce principals to each

other or to process transactions between them can give great savings, but if the third parties ever become untrustworthy then it can impose significant costs.

Another lesson is that if a component that fails (or can be induced to fail by an opponent) gives the wrong answer rather than just no answer, then it's much harder to build a resilient system using it. This has recently become a problem in avionics, leading to an emergency Airworthiness Directive in April 2005 that mandated a software upgrade for the Boeing 777, after one of these planes suffered a 'flight control outage' [762].

6.3.1.2 Interaction with Fault Tolerance

We can constrain the failure rate in a number of ways. The two most obvious are by using *redundancy* and *fail-stop processors*. The latter process error-correction information along with data, and stop when an inconsistency is detected; for example, bank transaction processing will typically stop if an out-of-balance condition is detected after a processing task. The two may be combined; IBM's System/88 minicomputer had two disks, two buses and even two CPUs, each of which would stop if it detected errors; the fail-stop CPUs were built by having two CPUs on the same card and comparing their outputs. If they disagreed the output went open-circuit, thus avoiding the Byzantine failure problem.

In general distributed systems, either redundancy or fail-stop processing can make a system more *resilient*, but their side effects are rather different. While both mechanisms may help protect the integrity of data, a fail-stop processor may be more vulnerable to service denial attacks, whereas redundancy can make confidentiality harder to achieve. If I have multiple sites with backup data, then confidentiality could be broken if any of them gets compromised; and if I have some data that I have a duty to destroy, perhaps in response to a court order, then purging it from multiple backup tapes can be a headache.

It is only a slight simplification to say that while replication provides integrity and availability, tamper resistance provides confidentiality too. I'll return to this theme later. Indeed, the prevalence of replication in commercial systems, and of tamper-resistance in military systems, echoes their differing protection priorities.

However, there are traps for the unwary. In one case in which I was called on as an expert, my client was arrested while using a credit card in a store, accused of having a forged card, and beaten up by the police. He was adamant that the card was genuine. Much later, we got the card examined by VISA who confirmed that it was indeed genuine. What happened, as well as we can reconstruct it, was this. Credit cards have two types of redundancy on the magnetic strip — a simple checksum obtained by combining together all the bytes on the track using exclusive-or, and a cryptographic checksum which we'll describe in detail later in section 10.5.2. The former is there to

detect errors, and the latter to detect forgery. It appears that in this particular case, the merchant's card reader was out of alignment in such a way as to cause an even number of bit errors which cancelled each other out by chance in the simple checksum, while causing the crypto checksum to fail. The result was a false alarm, and a major disruption in my client's life.

Redundancy is hard enough to deal with in mechanical systems. For example, training pilots to handle multi-engine aircraft involves drilling them on engine failure procedures, first in the simulator and then in real aircraft with an instructor. Novice pilots are in fact more likely to be killed by an engine failure in a multi-engine plane than in a single; landing in the nearest field is less hazardous for them than coping with suddenly asymmetric thrust. The same goes for instrument failures; it doesn't help to have three artificial horizons in the cockpit if, under stress, you rely on the one that's broken. Aircraft are much simpler than many modern information systems — yet there are still regular air crashes when pilots fail to manage the redundancy that's supposed to keep them safe. All too often, system designers put in multiple protection mechanisms and hope that things will be 'all right on the night'. This might be compared to strapping a 40-hour rookie pilot into a Learjet and telling him to go play. It really isn't good enough. Please bear the aircraft analogy in mind if you have to design systems combining redundancy and security!

The proper way to do things is to consider all the possible use cases and abuse cases of a system, think through all the failure modes that can happen by chance (or be maliciously induced), and work out how all the combinations of alarms will be dealt with — and how, and by whom. Then write up your safety and security case and have it evaluated by someone who knows what they're doing. I'll have more to say on this later in the chapter on 'System Evaluation and Assurance'.

Even so, large-scale system failures very often show up dependencies that the planners didn't think of. For example, Britain suffered a fuel tanker drivers' strike in 2001, and some hospitals had to close because of staff shortages. The government allocated petrol rations to doctors and nurses, but not to schoolteachers. So schools closed, and nurses had to stay home to look after their kids, and this closed hospitals too. We are becoming increasingly dependent on each other, and this makes contingency planning harder.

6.3.2 What Is Resilience For?

When introducing redundancy or other resilience mechanisms into a system, we need to be very clear about what they're for. An important consideration is whether the resilience is contained within a single organization.

In the first case, replication can be an internal feature of the server to make it more trustworthy. AT&T built a system called *Rampart* in which a number

of geographically distinct servers can perform a computation separately and combine their results using threshold decryption and signature [1065]; the idea is to use it for tasks like key management [1066]. IBM developed a variant on this idea called *Proactive Security*, where keys are regularly flushed through the system, regardless of whether an attack has been reported [597]. The idea is to recover even from attackers who break into a server and then simply bide their time until enough other servers have also been compromised. The trick of building a secure 'virtual server' on top of a number of cheap off-the-shelf machines has turned out to be attractive to people designing certification authority services because it's possible to have very robust evidence of attacks on, or mistakes made by, one of the component servers [337]. It also appeals to a number of navies, as critical resources can be spread around a ship in multiple PCs and survive most kinds of damage that don't sink it [489].

But often things are much more complicated. A server may have to protect itself against malicious clients. A prudent bank, for example, will assume that some of its customers would cheat it given the chance. Sometimes the problem is the other way round, in that we have to rely on a number of services, none of which is completely trustworthy. Since 9/11, for example, international money-laundering controls have been tightened so that people opening bank accounts are supposed to provide two different items that give evidence of their name and address — such as a gas bill and a pay slip. (This causes serious problems in Africa, where the poor also need banking services as part of their path out of poverty, but may live in huts that don't even have addresses, let alone utilities [55].)

The direction of mistrust has an effect on protocol design. A server faced with multiple untrustworthy clients, and a client relying on multiple servers that may be incompetent, unavailable or malicious, will both wish to control the flow of messages in a protocol in order to contain the effects of service denial. So a client facing several unreliable servers may wish to use an authentication protocol such as the Needham-Schroeder protocol I discussed in section 3.7.2; then the fact that the client can use old server tickets is no longer a bug but a feature. This idea can be applied to protocol design in general [1043]. It provides us with another insight into why protocols may fail if the principal responsible for the design, and the principal who carries the cost of fraud, are different; and why designing systems for the real world in which everyone (clients and servers) are unreliable and mutually suspicious, is hard.

At a still higher level, the emphasis might be on *security renewability*. Pay-TV is a good example: secret keys and other subscriber management tools are typically kept in a cheap smartcard rather than in an expensive set-top box, so that even if all the secret keys are compromised, the operator can recover by mailing new cards out to his subscribers. I'll discuss in more detail in Chapter 22, 'Copyright and Privacy Protection'.

6.3.3 At What Level Is the Redundancy?

Systems may be made resilient against errors, attacks and equipment failures at a number of levels. As with access control systems, these become progressively more complex and less reliable as we go up to higher layers in the system.

Some computers have been built with redundancy at the hardware level, such as the IBM System/88 I mentioned earlier. From the late 1980's, these machines were widely used in transaction processing tasks (eventually ordinary hardware became reliable enough that banks would not pay the premium in capital cost and development effort to use non-standard hardware). Some more modern systems achieve the same goal with standard hardware either at the component level, using *redundant arrays of inexpensive disks* ('RAID' disks) or at the system level by massively parallel server farms. But none of these techniques provides a defense against faulty or malicious software, or against an intruder who exploits such software.

At the next level up, there is *process group redundancy*. Here, we may run multiple copies of a system on multiple servers in different locations, and compare their outputs. This can stop the kind of attack in which the opponent gets physical access to a machine and subverts it, whether by mechanical destruction or by inserting unauthorized software, and destroys or alters data. It can't defend against attacks by authorized users or damage by bad authorized software, which could simply order the deletion of a critical file.

The next level is *backup*. Here, we typically take a copy of the system (also known as a *checkpoint*) at regular intervals. The backup copies are usually kept on media that can't be overwritten such as write-protected tapes or DVDs. We may also keep *journals* of all the transactions applied between checkpoints. In general, systems are kept recoverable by a transaction processing strategy of logging the incoming data, trying to do the transaction, logging it again, and then checking to see whether it worked. Whatever the detail, backup and recovery mechanisms not only enable us to recover from physical asset destruction; they also ensure that if we do get an attack at the logical level — such as a time bomb in our software which deletes our customer database on a specific date — we have some hope of recovering. They are not infallible though. The closest that any bank I know of came to a catastrophic computer failure that would have closed its business was when its mainframe software got progressively more tangled as time progressed, and it just wasn't feasible to roll back processing several weeks and try again.

Backup is not the same as *fallback*. A fallback system is typically a less capable system to which processing reverts when the main system is unavailable. An example is the use of manual imprinting machines to capture credit card transactions from the card embossing when electronic terminals fail.

Fallback systems are an example of redundancy in the application layer — the highest layer we can put it. We might require that a transaction above a certain limit be authorized by two members of staff, that an audit trail be kept of all transactions, and a number of other things. We'll discuss such arrangements at greater length in the chapter on banking and bookkeeping.

It is important to realise that these are different mechanisms, which do different things. Redundant disks won't protect against a malicious programmer who deletes all your account files, and backups won't stop him if rather than just deleting files he writes code that slowly inserts more and more errors. Neither will give much protection against attacks on data confidentiality. On the other hand, the best encryption in the world won't help you if your data processing center burns down. Real world recovery plans and mechanisms can get fiendishly complex and involve a mixture of all of the above.

The remarks that I made earlier about the difficulty of redundancy, and the absolute need to plan and train for it properly, apply in spades to system backup. When I was working in banking we reckoned that we could probably get our backup system working within an hour or so of our main processing centre being destroyed, but the tests we did were limited by the fact that we didn't want to risk processing during business hours. The most impressive preparations I've ever seen were at a UK supermarket, which as a matter of policy pulls the plug on its main processing centre once a year without warning the operators. This is the only way they can be sure that the backup arrangements actually work, and that the secondary processing centre really cuts in within a minute or so. Bank tellers can keep serving customers for a few hours with the systems down; but retailers with dead checkout lanes can't do that.

6.3.4 Service-Denial Attacks

One of the reasons we want security services to be fault-tolerant is to make service-denial attacks less attractive, more difficult, or both. These attacks are often used as part of a larger attack plan. For example, one might swamp a host to take it temporarily offline, and then get another machine on the same LAN (which had already been subverted) to assume its identity for a while. Another possible attack is to take down a security server to force other servers to use cached copies of credentials.

A powerful defense against service denial is to prevent the opponent mounting a selective attack. If principals are anonymous — or at least there is no name service which will tell the opponent where to attack — then he may be ineffective. I'll discuss this further in the context of burglar alarms and electronic warfare.

Where this isn't possible, and the opponent knows where to attack, then there are some types of service-denial attacks which can be stopped by redundancy and resilience mechanisms, and others which can't. For example, the TCP/IP protocol has few effective mechanisms for hosts to protect themselves against various network flooding attacks. An opponent can send a large number of connection requests and prevent anyone else establishing a connection. Defense against this kind of attack tends to involve moving your site to a beefier hosting service with specialist packet-washing hardware — or tracing and arresting the perpetrator.

Distributed denial-of-service (DDoS) attacks had been known to the research community as a possibility for some years. They came to public notice when they were used to bring down Panix, a New York ISP, for several days in 1996. During the late 1990s they were occasionally used by script kiddies to take over chat servers. In 2000, colleagues and I suggested dealing with the problem by server replication [1366], and in 2001 I mentioned them in passing in the first edition of this book. Over the following three years, small-time extortionists started using DDoS attacks for blackmail. The modus operandi was to assemble a *botnet*, a network of compromised PCs used as attack robots, which would flood a target webserver with packet traffic until its owner paid them to desist. Typical targets were online bookmakers, and amounts of $10,000–$50,000 were typically demanded to leave them alone. The typical bookie paid up the first time this happened, but when the attacks persisted the first solution was replication: operators moved their websites to hosting services such as Akamai whose servers are so numerous (and so close to customers) that they can shrug off anything that the average botnet could throw at them. In the end, the blackmail problem was solved when the bookmakers met and agreed not to pay any more blackmail money, and the Russian police were prodded into arresting the gang responsible.

Finally, where a more vulnerable fallback system exists, a common technique is to force its use by a service denial attack. The classic example is the use of smartcards for bank payments in countries in Europe. Smartcards are generally harder to forge than magnetic strip cards, but perhaps 1% of them fail every year, thanks to static electricity and worn contacts. Also, foreign tourists still use magnetic strip cards. So card payment systems have a fallback mode that uses the magnetic strip. A typical attack nowadays is to use a false terminal, or a bug inserted into the cable between a genuine terminal and a branch server, to capture card details, and then write these details to the magnetic stripe of a card whose chip has been destroyed (connecting 20V across the contacts does the job nicely). In the same way, burglar alarms that rely on network connections for the primary response and fall back to alarm bells may be very vulnerable if the network can be interrupted by an attacker: now that online alarms are the norm, few people pay attention any more to alarm bells.