CHAPTER
6

Distributed Systems

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.
—LESLIE LAMPORT

We've seen in the last few chapters how people can authenticate themselves to systems (and systems can authenticate themselves to each other) using security protocols; how access controls can be used to manage which principals can perform which operations in a system; and some of the mechanics of how crypto can be used to underpin access control in distributed systems. But there's much more to building a secure distributed systems than just implementing access controls, protocols, and crypto. When systems become large, the scale-up problems are not linear; there is often a qualitative change in complexity, and some things that are trivial to deal with in a network of only a few machines and principals (such as naming) suddenly become a big deal.

Over the last 35 years, computer science researchers have built many distributed systems and studied issues such as concurrency, failure recovery, and naming. The theory is also supplemented by growing body of experience from industry, commerce, and government. These issues are central to the design of effective secure systems, but are often handled rather badly. I've already described attacks on security protocols that can be seen as concurrency failures. If we replicate data to make a system fault-tolerant, then we may increase the risk of a compromise of confidentiality. Finally, naming difficulties are probably the main impediment to the construction of public key infrastructures.

6.1 Concurrency

Processes are said to be *concurrent* if they run at the same time, and concurrency gives rise to a number of well-studied problems. Processes may use old data; they can make inconsistent updates; the order of updates may or may not matter; the system might deadlock; the data in different systems might never converge to consistent values; and when it's important to know the exact time, this can be harder than you might think.

Programming concurrent systems is a hard problem in general; and, unfortunately, most of the textbook examples come from the relatively rarefied world of operating system internals and thread management. But concurrency control is also a security issue; like access control, it exists in order to prevent users interfering with each other, whether accidentally or on purpose. Also, concurrency problems can occur at a number of levels in a system, from the hardware right up to the business environment. In what follows, I provide a number of concrete examples that illustrate the effects of concurrency on security. Of course, these are by no means exhaustive.

6.1.1 Using Old Data versus Paying to Propagate State

I've already described two kinds of concurrency problem. First, there are replay attacks on protocols, where an attacker manages to pass off out-of-date credentials. Second, there are race conditions. I mentioned the ~~mkdir~~ vulnerability from Unix, in which a privileged program that is executed in two phases can be attacked halfway through the process by renaming an object on which it acts. These problems have been around for a long time. In one of the first multiuser operating systems, IBM's OS/360, an attempt to open a file caused it to be read and its permissions checked; if the user was authorized to access it, it was read again. The user could arrange things so that the file was altered in between [493].

These are examples of a *time-of-check-to-time-of-use* (TOCTTOU) attack. (A systematic way of finding such attacks is presented in [107].) However, preventing them isn't always economical, as propagating changes in security state can be expensive.

For example, the banking industry manages lists of all *hot* credit cards (whether stolen or abused); but there are millions of them worldwide, so it isn't possible to keep a complete hot-card list in every merchant terminal, and it would be too expensive to verify all transactions with the bank that issued the card. Instead, there are multiple levels of stand-in processing. Terminals are allowed to process transactions up to a certain limit (the *floor limit*) offline; larger transactions need online verification with a local bank, which will know about all the local hot cards, plus foreign cards that are being actively abused; above another limit there might be a reference to an organization such as VISA with a larger international list; while the largest transactions might need a reference to the card issuer. In effect, the only transactions that are checked immediately before use are those that are local or large.

Credit cards are interesting because, as people start to build infrastructures of public key certificates to support Web shopping based on the SSL, and corporate networks based on Win2K, there's a fear that the largest cost will be the revocation of public key certificates belonging to principals whose credentials have changed—because they changed address, changed job, had their private key hacked, or whatever. Credit card networks are the largest existing systems that manage the global propagation of security state—which they do by assuming that most events are local, of low value, or both.

6.1.2 Locking to Prevent Inconsistent Updates

When a number of people are working concurrently on a document, they may use a product such as RCS to ensure that only one person has write access at any one time to any given part of it. This illustrates the importance of *locking* as a way to manage contention for resources, such as filesystems, and to reduce the likelihood of conflicting updates. Another mechanism is *callback;* a server may keep a list of all those clients that rely on it for security state, and notify them when the state changes.

These are also issues in secure distributed systems. Credit cards provide an example. If I own a hotel, and a customer presents a credit card on checkin, I ask the card company for a *preauthorization*, which records the fact that I will want to make a debit in the near future; I might register a claim on "up to $500" of her available credit. If the card is cancelled, the following day, her bank can call me and ask me to contact the police or to get her to pay cash. (My bank might or might not have guaranteed me the money; it all depends on the sort of contract I've managed to negotiate with it.) This is an example of the *publish-register-notify* model of how to do robust authorization in distributed systems (of which there's a more general description in [65]).

Callback mechanisms don't provide a universal solution, though. The credential issuer might not want to run a callback service, and the customer might object on privacy grounds to the issuer being told all her comings and goings. Consider passports, for example. In many countries, government ID is required for many transactions, but governments won't provide any guarantee, and most citizens would object if the government kept a record of every time a government-issue ID was presented.

In general, there is a distinction between those credentials whose use gives rise to some obligation on the issuer, such as credit cards, and the others, such as passports. Among the differences is the importance of the order in which updates are made.

6.1.3 Order of Updates

If two large transactions arrive at the government's bank account—say a credit of $500,000 and a debit of $400,000—then the order in which they are applied may not matter much. But if they're arriving at my bank account, the order will have a huge effect on the outcome! In fact, the problem of deciding the order in which transactions are applied has no clean solution. It's closely related to the problem of how to parallelize a computation, and much of the art of building efficient distributed systems lies in arranging matters so that processes are either simple sequential or completely parallel.

The usual algorithm in retail checking account systems is to batch the transctions overnight and apply all the credits for each account before applying all the debits. The inevitable side effect of this is that payments that bounce have to be reversed out. In practice, chains of failed payments terminate, though in theory this isn't necessarily so. In order to limit the *systemic risk* that a nonterminating payment revocation chain might bring down the world's banking system, some interbank payment mechanisms are moving to *real-time gross settlement* (RTGS), whereby transactions are booked in order of arrival. The downside here is that the outcome can depend on network vagaries. Credit cards operate a mixture of the two strategies, with credit limits run in real time or near real time (each authorization reduces the available credit limit), while settlement is run just as in a checking account. The downside of this is that by putting through a large preauthorization, a merchant can tie up your card.

The checking account approach has recently been the subject of research in the parallel systems community. The idea is that disconnected applications propose tentative update transactions that are later applied to a master copy. Various techniques can be used to avoid instability; mechanisms for tentative update, such as with bank journals, are particularly important [352].

In other systems, the order in which transactions arrive is much less important. Passports are a good example. Passport issuers only worry about their creation and expiration dates, not the order in which visas are stamped on them.

6.1.4 Deadlock

Deadlock is another problem. Things may foul up because two systems are each waiting for the other to move first. A famous exposition of deadlock is the *dining philosophers' problem.* A number of philosophers are seated round a table; each has a chopstick on his left, and can eat only when he can pick up the two chopsticks on either side. Deadlock can follow if they all try to eat at once, and each picks up, say, the chopstick on his right. (This problem, and the algorithms that can be used to avoid it, are presented in a classic paper by Dijkstra [251].)

This can get horribly complex when you have multiple hierarchies of locks, and they're distributed across systems, some of which fail (especially where failures can mean that the locks aren't reliable). A lot has been written on the problem in the distributed systems literature [64]. But it is not just a technical matter; there are many catch-22 situations in business processes. As long as the process is manual, some fudge may be found to get round the catch, but when it is implemented in software, this option may no longer be available.

Sometimes it isn't possible to remove the fudge. In a well-known problem in business—the *battle of the forms*—one company issues an order with its own terms attached, another company accepts it subject to its own terms, and trading proceeds without any agreement about whose conditions govern the contract. This promises to worsen as trading becomes more electronic.

6.1.5 Non-convergent State

When designing protocols that update the state of a distributed system, the conventional wisdom is ACID—transactions should be *atomic, consistent, isolated, and durable.* A transaction is atomic if you "do it all or not at all"—which makes it easier to recover the system after a failure. It is consistent if some invariant is preserved, such as that the books must still balance. This is common in banking systems, and is achieved by insisting that each credit to one account is matched by an equal and opposite debit to another (I discuss this more in Chapter 9, "Banking and Bookkeeping"). Transactions are isolated if they look the same to each other, that is, are serializable; and they are durable if once done they can't be undone.

These properties can be too much, or not enough, or both. Each of them can fail or be attacked in numerous obscure ways, and it's often sufficient to design the system to be *convergent.* This means that, if the transaction volume were to tail off, then eventually there would be consistent state throughout [565]. Convergence is usually achieved using semantic tricks such as timestamps and version numbers; it can often be enough where transactions get appended to files rather than overwritten.

However, in real life, there must also be ways to survive things that go wrong and that are not completely recoverable. The life of a security or audit manager can be a constant battle against entropy: apparent deficits (and surpluses) are always turning up, and sometimes simply can't be explained. For example, different national systems have different ideas of which fields in bank transaction records are mandatory or optional, so payment gateways often have to guess data in order to make things work. Sometimes they guess wrong; and sometimes people see and exploit vulnerabilities that aren't understood until much later (if ever). In the end, things get fudged by adding a correction factor, called something like "branch differences," and setting a target for keeping it below a certain annual threshold.

The battle of the forms just mentioned gives an example of a distributed nonelectronic system that doesn't converge.

In military systems, there is the further problem of dealing with users who request some data for which they don't have a clearance. For example, someone might ask the destination of a warship that's actually on a secret mission carrying arms to Iran. If the user isn't allowed to know this, the system may conceal the fact that the ship is doing something secret by making up a *cover story.* (The problems this causes will be discussed at greater length in Chapter 7, "Multilevel Security.")

6.1.6 Secure Time

The final kind of concurrency problem with special interest to the security engineer is the provision of accurate time. As authentication protocols such as Kerberos can be attacked by inducing an error in the clock, it's not enough to simply trust a time source on the network. There is a dangerous recursion in relying exclusively on secure time for network authentication, as the master clock signal must itself be authenticated. One of the many bad things that can happen if this isn't done right is a *Cinderella attack.* If a security-critical program such as a firewall has a license with a timelock in it, a bad man (or a virus) could wind your clock forward "and cause your software to turn into a pumpkin."

There are several possible approaches:

- 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 designed to make clock failures and network delays apparent. Even though these are designed to withstand random (rather than malicious) failure, they can often be hardened by having the messages digitally signed.
- You can abandon absolute time and instead use *Lamport time*, which means that all you care about is whether event A happened before event B, rather than what date it is [486]. 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 [364].

In most applications, you may 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.

6.2 Fault Tolerance and Failure Recovery

Failure recovery is often the most important aspect of security engineering, yet is one of the most neglected. For many years, 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 to availability and recovery mechanisms, such as hot standby processing sites and multiply redundant networks; a few percent are invested in integrity mechanisms such as internal audit; and an almost insignificant amount gets spent 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 system fault tolerance is usually based on mechanisms such as logs and locking, and is greatly complicated when it must be made resilient in the face of malicious attacks on these mechanisms. It 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 that 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.

6.2.1 Failure Models

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

6.2.1.1 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. Each general can exchange confidential and authentic communications with each other general (we can also imagine them encrypting and computing a MAC on each message). What is the maximum number t of traitors that 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 be 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 Lamport, Shostack, and Peace, who prove that the problem has a solution if and only if $n \geq 3t + 1$ [487]. 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.

6.2.1.2 Interaction with Fault Tolerance

We can constrain the failure rate in a number of ways. The two most obvious are by using *fail-stop processors* and *redundancy*. Either of these can make the system more *resilient*, but their side effects are rather different. Briefly, while both mechanisms may be effective at protecting the integrity of data, a fail-stop processor is likely to be more vulnerable to service denial attacks, whereas redundancy makes 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 backup tapes can be a nightmare.

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.

Still, 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 all the bytes on the track using exclusive-or, and a cryptographic checksum, which I’ll describe in detail later in Section 19.3.2. The former is there to detect errors, 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.

6.2.2 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 has 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 [639]; the idea is to use it for tasks such as key management [640]. IBM has a variant on this idea called Proactive Security. Here, keys are regularly flushed through the system, regardless of whether an attack has been reported [379]. 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 [211]. 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 actually sink the vessel [309].

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 many 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. In countries without national ID card systems, for example, a retailer who wants to extend credit to a customer may ask to see three different items that give evidence of the customer's name and address (say, a gas bill, a phone bill, and a pay slip).

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. Thus, a client facing several unreliable servers may wish to use an authentication protocol, such as the Needham-Schroeder protocol discussed above; there, 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 [623]. 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, where all principals 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 the subscribers. I'll go into this in more detail in Chapter 20, "Copyright and Privacy Protection."

6.2.3 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 multiple CPUs and mirrored disks, to reduce the probability of failure. From the late 1980s, these machines were widely used in transaction processing tasks. Some more modern systems achieve the same goal using massively parallel server farms; *redundant arrays of inexpensive disks* (RAID disks) are a similar concept. But none of these techniques provides a defense against an intruder, let alone faulty or malicious software.

At the next level up is *process group redundancy*. Here, we may run multiple copies of a system on multiple servers in different locations, and get them to vote on the output. 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.

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 tapes with the write-protect tab set, or CDs. We may also keep *journals* of all the transactions applied between checkpoints. In general, systems are made 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 suffer an attack at the logical level—such as a time bomb in our software that deletes our customer database on a specific date—we have some hope of recovering. These mechanisms are not infallible, though. The closest that any bank I know of came to a catastrophic computer failure that would have closed their business was when their 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 "zip-zap" machines to capture credit card transactions when electronic terminals fail.

Fallback systems are an example of redundancy in the application layer—the highest layer where 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. I'll discuss such arrangements at greater length in Chapter 9.

It is important to realize that hardware redundancy, group redundancy, backup and fallback 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.

6.2.4 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, 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 very 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 that will tell the opponent where to attack—then an attack 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, some types of service denial attacks can be stopped by redundancy and resilience mechanisms, and others 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, to prevent anyone else establishing a connection. Defense against this kind of attack tends to involve tracing and arresting the perpetrator.

Recently, there has been software on the Net that helps the opponent to hack a number of undefended systems and use these as attack robots to flood the victim. I'll discuss this in Chapter 18, "Network Attack and Defense." For now, I'll just remark that stopping such attacks is hard, and replication isn't a complete solution. If you just switch to a backup machine, and tell the name service, it will happily give the new IP address to the attack software as well as to everybody else. Where such a strategy may be helpful is if the backup machine is substantially more capable and thus can cope better with the load. For example, you might failover to a high-capacity Web hosting service. This is in some sense the opposite concept to "fallback."

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 such as France and Norway. Smartcards are generally harder to forge than magnetic strip cards, but perhaps 1 percent of them fail every year, thanks to environmental insults such as static. Also, foreign tourists still use magnetic strip cards. So smartcard payment systems need a fallback mode that does traditional processing. Many attacks target this fallback mode. One trick is to destroy a smartcard chip by connecting it to the electricity mains; a more common trick is just to use credit cards stolen from foreign tourists, or imported from criminals in countries where magnetic stripes are still the norm. In the same way, burglar alarms that rely on network connections for the primary response and fallback to alarm bells may be very vulnerable if the network can be interrupted by an attacker. Few people pay attention any more to alarm bells.

6.3 Naming

Naming is a minor, if troublesome, aspect of ordinary distributed systems, but it becomes surprisingly hard in security engineering. A topical example (as of 2000) is the problem of what sort of names to put on public key certificates. A certificate that says simply, "The person named Ross Anderson is allowed to administer system X" is of little use. Before the arrival of Internet search engines, I was the only Ross Anderson I knew of; now I know of dozens of us. I am also known by different names to dozens of different systems. Names exist in contexts, and naming the principals in secure systems is becoming ever more important and difficult.

There is some hope. Most (though not all) of the problems encountered so far have come from ignoring the established lessons of naming in ordinary distributed systems.

6.3.1 The Distributed Systems View of Naming

During the last quarter of the twentieth century, the distributed systems research community ran up against many naming problems. The basic algorithm used to bind names to addresses is known as *rendezvous:* the principal exporting a name advertises it somewhere, and the principal seeking to import and use it searches for it. Obvious examples include phone books and directories in file systems.

However, the distributed systems community soon realized that naming can get fiendishly complex, and the lessons learned are set out in a classic article by Needham [587]. I'll summarize the main points, and look at which of them apply to secure systems.

1. *The function of names is to facilitate sharing.* This continues to hold: my bank account number exists in order to provide a convenient way of sharing the information that I deposited money last week with the teller from whom I am trying to withdraw money this week. In general, names are needed when the data to be shared is changeable. If I only ever wished to withdraw exactly the same sum as I'd deposited, a bearer deposit certificate would be fine. Conversely, names need not be shared—or linked—where data will not be; there is no need to link my bank account number to my telephone number unless I am going to pay my phone bill from the account.
2. *The naming information may not all be in one place, so resolving names brings all the general problems of a distributed system.* This holds with a vengeance. A link between a bank account and a phone number assumes both of them will remain stable. When each system relies on the other, an attack on one can affect both. In the days when electronic banking was dial-up rather than Web-based, a bank that identified its customers using calling-line ID was vulnerable to attacks that circumvented the security of the telephone exchange (such as tapping into the distribution frame in an apartment block, hacking a phone company computer, or bribing a phone company employee).
3. *It is bad to assume that only so many names will be needed.* The shortage of IP addresses, which motivated the development of IP version 6 (IPv6), is well enough discussed. What is less well known is that the most expensive upgrade that the credit card industry ever had to make was not Y2K remediation, but the move from 13-digit credit card numbers to 16. Issuers originally assumed that 13 digits would be enough; but the system ended up with tens of thousands of banks (many with dozens of products), so a 6-digit *bank identification number* (BIN number) was needed. Some card issuers have millions of

customers, so a 9-digit account number is the norm. And there's also a *check digit* (a linear combination of the other digits, which is appended to detect errors).

4. *Global names buy you less than you think.* For example, the 128-bit addresses planned for IPv6 can enable every object in the universe to have a unique name. However, for us to do business, a local name at my end must be resolved into this unique name and back into a local name at your end. Invoking a unique name in the middle may not buy us anything; it may even get in the way if the unique naming service takes time, costs money, or occasionally fails (as it surely will). In fact, the name service itself will usually have to be a distributed system, of the same scale (and security level) as the system we're trying to protect. So we can expect no silver bullets from this quarter. One reason the banking industry is wary of initiatives to set up public key infrastructures which would give each citizen the electronic equivalent of an ID card, is that banks already have unique names for their customers (account numbers). Adding an extra number does little good, but it has the potential to add extra costs and failure modes.
5. *Names imply commitments, so keep the scheme flexible enough to cope with organizational changes.* This sound principle was ignored in the design of Cloud Cover, the U.K. government's key management system for secure email [50]. There, principals' private keys are generated by encrypting their names under departmental master keys. So reorganizations mean that the security infrastructure must be rebuilt.
6. *Names may double as access tickets, or capabilities.* We have already seen a number of examples of this in the chapters on protocols and passwords. In general, it's a bad idea to assume that today's name won't be tomorrow's password or capability—remember the Utrecht fraud discussed in Section 2.4. (This is one of the arguments for making all names public keys—"keys speak in cyberspace" in Carl Ellison's phrase—but we've already noted the difficulties of linking keys with names.)

   I've given a number of examples of how things go wrong when a name starts being used as a password. But sometimes the roles of name and password are ambiguous. In order to get entry to the car park I use at the university, I speak my surname and parking badge number into a microphone near the barrier. So if I say, "Anderson, 123" (or whatever), which of these is the password? (In fact it's "Anderson," as anyone can walk through the car park and note down valid badge numbers from the parking permits displayed on the cars.) In this context, a lot deserves to be said about biometrics, which I'll postpone until Chapter 13.
7. *Things are made much simpler if an incorrect name is obvious.* In standard distributed systems, this enables us to take a liberal attitude toward cacheing. In payment systems, credit card numbers may be accepted while a terminal is offline as long as the credit card number appears valid (i.e., the last digit is a proper check digit of the first 15) and is not on the hot-card list. Certificates provide a higher-quality implementation of the same basic concept.

It's important where the name is checked. The credit card check digit algorithm is deployed at the point of sale, so it is inevitably public. A further check—the *card verification value* (CVV) on the magnetic strip—is computed with secret keys, but can be checked at the issuing bank, the acquiring bank, or even at a network switch (if one trusts these third parties with the keys). This is more expensive, and still vulnerable to network outages.

8. *Consistency is hard, and is often fudged. If directories are replicated, then you may find yourself unable to read, or to write, depending on whether too many or too few directories are available.* Naming consistency causes problems for e-commerce in a number of ways, of which perhaps the most notorious is the barcode system. Although this is simple enough in theory—with a unique numerical code for each product—in practice, it can be a nightmare, as different manufacturers, distributors, and retailers attach quite different descriptions to the barcodes in their databases. Thus, a search for products by "Kellogg's" will throw up quite different results depending on whether or not an apostrophe is inserted, and this can cause great confusion in the supply chain. Proposals to fix this problem can be surprisingly complicated [387].

   There are also the issues of covergence discussed above; data might not be consistent across a system, even in theory. There are also the problems of timeliness, such as the revocation problem for public key certificates.

9. *Don't get too smart. Phone numbers are much more robust than computer addresses.* Amen to that; but it's too often ignored by secure system designers. Bank account numbers are much easier to deal with than the X.509 certificates proposed for protocols such as SET—which was supposed to be the new standard for credit card payments on the Net, but which has so far failed to take off as a result of its complexity and cost. I discuss X.509 and SET in Part 2.

10. *Some names are bound early, others not; and in general it is a bad thing to bind early if you can avoid it.* A prudent programmer will normally avoid coding absolute addresses or filenames, as that would make it hard to upgrade or replace a machine. He will prefer to leave this to a configuration file or an external service such as DNS. (This is another reason not to put addresses in names.) Here, there can be a slight tension with some protection goals: secure systems often want stable and accountable names, as any third-party service used for last-minute resolution could be a point of attack. Knowing them well in advance permits preauthorization of transactions and other such optimizations.

So, of Needham's 10 principles for distributed naming, nine apply directly to distributed secure systems. The (partial) exception is whether names should be bound early or late.

6.3.2 What Else Goes Wrong

Needham's principles, although very useful, are not sufficient. They were designed for a world in which naming systems could be designed and imposed at the system owner's convenience. When we move from distributed systems in the abstract to the reality of modern Internet-based (and otherwise interlinked) service industries, there is quite a lot more to say.

6.3.2.1 Naming and Identity

The most obvious difference is that the principals in security protocols may be known by many different kinds of name—a bank account number, a company registration number, a personal name plus a date of birth or a postal address, a telephone number, a passport number, a health service patient number, or a userid on a computer system.

As I mentioned in the introductory definitions, a common mistake is to confuse naming with identity. *Identity* is when two different names (or instances of the same name) correspond to the same principal (this is known in the distributed systems literature as an *indirect name* or *symbolic link*). The classic example comes from the registration of title to real estate. It is very common that someone who wishes to sell a house uses a different name than they did at the time it was purchased: they might have changed name on marriage, or after a criminal conviction. Changes in name usage are also common. For example, the DE Bell of the Bell-LaPadula system (which I'll discuss in the next chapter) wrote his name "D. Elliot Bell" in 1973 on that paper; but he was always known as David, which is how he now writes his name, too. A land registration system must cope with a lot of identity issues like this.

A more typical example of identity might be a statement such as, "The Jim Smith who owns bank account number 12345678 is the Robert James Smith with passport number 98765432 and date of birth 3/4/56." It may be seen as a symbolic link between two separate systems—the bank's and the passport office's. Note that the latter part of this identity encapsulates a further identity, which might be something like, "The U.S. passport office's file number 98765432 corresponds to the entry in birth register for 3/4/56 of one Robert James Smith." In general, names may involve several steps of recursion.

6.3.2.2 Cultural Assumptions

The assumptions that underlie names often change from one country to another. In the English-speaking world, people may generally use as many names as they please; a name is simply what you are known by. But some countries forbid the use of aliases, and others require them to be registered. This can lead to some interesting scams. In at least one case, a British citizen has evaded pursuit by foreign tax authorities by changing his name. On a less exalted plane, women who pursue academic careers and change their name on marriage may wish to retain their former name for professional use, which means that the name on their scientific papers is different from their name on the payroll. This has caused a huge row at my university, which introduced a unified ID card system keyed to payroll names, without support for aliases.

In general, many of the really intractable problems arise when an attempt is made to unify two local naming systems that turn out to have incompatible assumptions. As electronics invade everyday life more and more, and systems become linked up, conflicts can propagate and have unexpected remote effects. For example, one of the lady professors in dispute over our university card is also a trustee of the British Library, which issues its own admission tickets on the basis of the name on the holder's home university library card.

Even human naming conventions are not uniform. Russians are known by a forename, a patronymic, and a surname; Icelanders have no surname but are known instead by a given name, followed by a patronymic if they are male and a matronymic if they are female. This causes problems when they travel. When U.S. immigration comes across Maria Trosttadóttir and learns that Trosttadóttir isn't a surname or even a patronymic, its standard practice is to compel her to adopt as a surname a patronymic (say, Carlsson if her father was called Carl). This causes unnecessary offense.

The biggest cultural divide is often thought to be that between the English-speaking countries, where identity cards are considered to be unacceptable on privacy grounds (unless they're called drivers' licenses or health service cards), and the countries conquered by Napoleon (or by the Soviets) where identity cards are the norm. Other examples are more subtle. I know Germans who refuse to believe that a country can function at all without a proper system of population registration and ID cards, yet are asked for their ID card only rarely (for example, to open a bank account or get married). Their card number can't be used as a name, because it is a document number and changes every time a new card is issued. A Swiss hotelier may be happy to register a German guest on sight of an ID card rather than a credit card, but if he discovers some damage after a German guest has checked out, he may be out of luck. And the British passport office will issue a citizen with more than one passport at the same time, if he says he needs them to make business trips to (say) Cuba and the USA; so our Swiss hotelier, finding that a British guest has just left without paying, can't rely on the passport number to have him stopped at the airport.

There are many other hidden assumptions about the relationship between governments and people's names, and they vary from one country to another in ways which can cause subtle security failures.

6.3.2.3 Semantic Content of Names

Another hazard arises on changing from one type of name to another without adequate background research. A bank got sued after it moved from storing customer data by account number to storing it by name and address. The bank wanted to target junk mail more accurately, so it had a program written to link all the accounts operated by each of its customers. The effect for one customer was that the bank statement for the account he maintained for his mistress got sent to his wife, who divorced him.

Sometimes naming is simple, but sometimes it merely appears to be. For example, when I got a monthly ticket for the local swimming pool, the cashier simply took the top card off a pile, swiped it through a reader to tell the system it was now live, and gave it to me. I had been assigned a random name—the serial number on the card. Many U.S. road toll systems work in much the same way. Sometimes a random, anonymous name can add commercial value. In Hong Kong, toll tokens for the Aberdeen tunnel could be bought for cash or at a discount in the form of a refillable card. In the run-up to the transfer of power from Britain to Beijing, many people preferred to pay extra for the less traceable version, as they were worried about surveillance by the new police force.

Semantics of names can change. I once got a hardware store loyalty card with a random account number (and no credit checks). I was offered the chance to change this into a bank card after the store was taken over by the supermarket, and the supermarket started a bank. (This appears to have ignored money-laundering regulations that all new bank customers must be identified and have references taken up.)

Assigning bank account numbers to customers might have seemed unproblematic—but as the above examples show, systems may start to construct assumptions about relationships between names that are misleading and dangerous.

6.3.2.4 Uniqueness of Names

Human names evolved when we lived in small communities. They were not designed for the Internet. There are now many more people (and systems) online than we are used to dealing with. As I remarked at the beginning of this section, I used to be the only Ross Anderson I knew of, but thanks to Internet search engines, I now know dozens of namesakes. Some of them work in fields I've also worked in, such as software engineering and electric power distribution; the fact that I'm ~~www. ross-anderson.com~~ and ~~ross.anderson@iee.org~~ is just luck—I got there first. (Even so, ~~rjanderson @iee.org~~ is somebody else.) So even the combination of a relatively rare name and a specialized profession is still ambiguous.

6.3.2.5 Stability of Names and Addresses

Many names include some kind of address, yet addresses change. About a quarter of Cambridge phone book addresses change every year; with email, the turnover is probably higher. A project to develop a directory of people who use encrypted email, together with their keys, found that the main cause of changed entries was changes of email address [42]. (Some people had assumed it would be the loss or theft of keys; the contribution from this source was precisely zero.)

A potentially serious problem could arise with IPv6. The security community assumes that v6 IP addresses will be stable, so that public key infrastructures can be built to bind principals of various kinds to them. All sorts of mechanisms have been proposed to map real-world names, addresses, and even document content indelibly and eternally onto 128-bit strings (see, for example, [365]). The data communications community, on the other hand, assumes that IPv6 addresses will change regularly. The more significant bits will change to accommodate more efficient routing algorithms, while the less significant bits will be used to manage local networks. These assumptions can't both be right.

Distributed systems pioneers considered it a bad thing to put addresses in names [565]. But in general, there can be multiple layers of abstraction, with some of the address information at each layer forming part of the name at the layer above. Also, whether a namespace is better flat depends on the application. Often people end up with different names at the departmental and organizational level (such as ~~rja14@cam.ac.uk~~ and ~~ross.anderson@cl.cam.ac.uk~~ in my own case). So a clean demarcation between names and addresses is not always possible.

Authorizations have many (but not all) of the properties of addresses. Designers of public key infrastructures are beginning to realize that if a public key certificate contains a list of what it may be used for, then the more things on this list the shorter its period of usefulness. A similar problem besets systems where names are composite. For example, some online businesses recognize me by the combination of email address and credit card number. This is clearly bad practice. Quite apart from the fact that I have several email addresses, I have several credit cards. The one I use will depend on which of them is currently giving the best cashback or the most air miles. (So if the government passes a law making the use of pseudonyms on the Net illegal, does this mean I have to stick to the one ISP and the one credit card?)

6.3.2.6 Restrictions on the Use of Names

This brings us to a further problem. Some names may be used only in restricted circumstances. This may be laid down by law, as with the U.S. *Social Security number* (SSN) and its equivalents in many European countries. Sometimes it is a matter of marketing. I would rather not give out my residential address (or my home phone number) when shopping on the Web, and will avoid businesses that demand them.

Memorable pseudonyms are sometimes required. In a university, one occasionally has to change email addresses, for example, when a student is a victim of cyberstalking. Another example is where a celebrity wants a private mailbox as well as the "obvious" one that goes to her secretary.

Sometimes it's more serious. Pseudonyms are often used as a *privacy-enhancing technology*. They can interact with naming in unexpected ways. For example, it's fairly common for hospitals to use a patient number as an index to medical record databases, as this may allow researchers to use pseudonymous records for some limited purposes without much further processing. This causes problems when a merger of health maintenance organizations, or a new policy directive in a national health service, forces the hospital to introduce uniform names. Patient confidentiality can be completely undermined. (I'll discuss anonymity further in Chapter 20, and its particular application to medical databases in Chapter 8.)

Finally, when we come to law and policy, the definition of a name turns out to be unexpectedly tricky. Regulations that allow police to collect communications data—that is, a record of who called whom and when—are often very much more lax than the regulations governing phone tapping; in many countries, police can get this data just by asking the phone company. An issue that caused a public outcry in the United Kingdom was whether this enables them to harvest the URLs that people use to fetch Web pages. URLs often have embedded in them data such as the parameters passed to search engines. Clearly, there are policemen who would like a list of everyone who hit a URL such as ~~`http://www.google.com/search?q=cannabis+cultivation+UK;`~~ just as clearly, many people would consider such large-scale trawling to be an unacceptable invasion of privacy. On the other hand, if the police are limited to monitoring IP addresses, they could have difficulties tracing criminals who use transient IP addresses provided by free ISP services.

6.3.3 Types of Name

The complexity is organizational and technical, as well as political. I noted in the introduction that names can refer not just to persons and machines acting on their behalf, but also to organizations, roles ("the officer of the watch"), groups, and compound constructions: *principal in role*—Alice as manager; *delegation*—Alice for Bob; *conjunction*—Alice and Bob. Conjunction often expresses implicit access rules: "Alice acting as branch manager plus Bob as a member of the group of branch accountants."

That's only the beginning. Names also apply to services (such as NFS or a public key infrastructure) and channels (which might mean wires, ports, or crypto keys). The same name might refer to different roles: "Alice as a computer game player" ought to have less privilege than "Alice the system administrator." The usual abstraction used in the security literature is to treat them as different principals. This all means that there's no easy mapping between names and principals.

Finally, there are functional tensions that come from the underlying business processes rather from system design. Businesses mainly want to get paid, while governments want to identify people uniquely. In effect, business wants a credit card number while government wants a passport number. Building systems that try to be both—as some governments are trying to encourage—is a tar-pit. There are many semantic differences. You can show your passport to a million people, if you wish, but you had better not try that with a credit card. Banks want to open accounts for anyone who turns up with some money; governments want them to verify people's identity carefully in order to discourage money laundering. The list is a long one.

6.4 Summary

Many secure distributed systems have incurred huge costs or developed serious vulnerabilities, because their designers ignored the basic lessons of how to build (and how not to build) distributed systems. Most of these lessons are still valid, and there are more to add.

A large number of security breaches are concurrency failures of one kind or another; systems use old data, make updates inconsistently or in the wrong order, or assume that data are consistent when they aren't and can't be. Knowing the right time is harder than it seems.

Fault tolerance and failure recovery are critical. Providing the ability to recover from security failures, and random physical disasters, is the main purpose of the protection budget for many organizations. At a more technical level, there are significant interactions between protection and resilience mechanisms. Byzantine failure—where defective processes conspire, rather than failing randomly—is an issue, and interacts with our choice of cryptographic tools. There are many different flavors of redundancy, and we have to use the right combination. We need to protect not just against failures and attempted manipulation, but also against deliberate attempts to deny service, which may often be part of larger attack plans.

Many problems also arise from trying to make a name do too much, or making assumptions about it which don't hold outside of one particular system, or culture, or jurisdiction. For example, it should be possible to revoke a user's access to a system by cancelling their user name without getting sued on account of other functions being revoked. The simplest solution is often to assign each principal a unique identifier used for no other purpose, such as a bank account number or a system logon name. But many problems arise when merging two systems that use naming schemes that are incompatible for some reason. Sometimes this merging can even happen by accident—an example being when two systems use a common combination such as "name plus date of birth" to track individuals.

Research Problems

In the research community, secure distributed systems tend to have been discussed as a side issue by experts on communications protocols and operating systems, rather than as a discipline in its own right. So it is a relatively open field, and one that I feel holds much promise over the next five to ten years.

There are many technical issues which I've touched on in this chapter, such as how we design secure time protocols and the complexities of naming. But perhaps the most important research problem is to work out how to design systems that are resilient in the face of malice, that degrade gracefully, and whose security can be recovered simply once the attack is past. This may mean revisiting the definition of convergent applications. Under what conditions can we recover neatly from corrupt security state? Do we have to rework recovery (which explores how to rebuild databases from backup tapes)? What interactions are there between recovery mechanisms and particular protection technologies? In what respects should protection mechanisms be separated from resilience mechanisms, and in what respects should they be separated? What other pieces are missing from the jigsaw?

Further Reading

There are many books on distributed systems. I've found Sape Mullender's anthology [565] to be helpful and thought-provoking for graduate students, while the textbook we recommend to our undergraduates by Jean Bacon [64] is also worth reading. Geraint Price has a survey of the literature on the interaction between fault tolerance and security [623]. The research literature on concurrency, such as the SIGMOD conferences, has occasional gems. But the most important practical topic for the working security engineer is probably contingency planning. There are many books on this topic; the one I have on my shelf is by Jon Toigo [749].