# Distributed Systems Goals & Challenges

Before we start discussing distributed systems architectures it is important to understand why we have been driven to build distributed systems, and the fundamental problems associated with doing so.

## Goals: Why Build Distributed Systems

Distributed systems could easily be justified by the simple facts of collaboration and sharing. The world-wide web is an obvious and compelling example of the value that is created when people can easily expose and exchange information. Much of the work we do is done in collaboration with others, and so we need often need to share work products. But there are several other powerful forces driving us towards distributed systems.

### Client/Server Usage Model

Long ago, all the equipment I needed to use in connection with my computer was connected to my computer. But this does not always make sense:

- I may only use a high resolution color scanner for a few minutes per month.
- I may make regular use of a high speed color printer, but not enough to justify buying one just for myself.
- I could store my music and videos on my own computer, but if I store them on a home NAS server, the entire family can have access to our combined libraries.
- I could store all my work related files on my own computer, but if I store them on a work-group server, somebody else will manage the back-ups, and ensure that everyone on the project has access to them.

There are many situations where we can get better functionality and save money by using remote/centralized resources rather than requiring all resources to be connected to a client computer.

### Reliability and Availability

As we come to depend more and more on our digital computers and storage we require higher reliability and availability from them. Long ago people tried to improve reliability by building systems out of the best possible components. But, as with RAID, we have learned that we can obtain better reliability by combining multiple (very ordinary systems). If we have more computing and storage capacity than we actually need, we may be able to continue providing service, even after one or more of our servers have failed.

The key is to distribute service over multiple independent servers. The reason they must be independent is so that they have no *single point of failure* ... no single component whose failure would take out multiple systems. If the client and server instances are to be distributed across multiple independent computers, then we are building a distributed system.

### Scalability

It is common to start any new project on a small system. If the system is successful, we will probably add more work to it over time. This means we will need more storage capacity, more network bandwidth, and more computing power. System manufacturers would be delighted if, each time we needed more capacity and power, we bought a new (larger, more expensive) computer (and threw away the old one). But

a. This is highly inefficient, as we are essentially throwing away old capacity in order to buy new capacity.
b. If we are successful, our needs for capacity and power will eventually exceed even the largest computer.

A more practical approach would be to design systems that can be expanded incrementally, by adding additional computers and storage as they were needed. And, again, if our growth plan is to *scale-out* (rather than *scale-up*) we are going to be building our system out of multiple independent computers, and so we are building a distributed system.

### Flexibility

We may start building and testing all the parts of a new service on a notebook or desktop, but later we may decide that we need to run different parts on different computers, or a single part on multiple computers. If our the components of our service interact with one-another through network protocols, it will likely be very easy to change the deployment model (which services run on which computers). Distributed systems tend to be very flexible in this respect.

## Challenges: Why are Distributed Systems Hard to Build

The short answer is that there are two reasons:

- Many solutions that work on single systems, do not work in distributed systems.
- Distributed systems have new problems that were never encountered in single systems.

### New and More Modes of Failure

If something bad happens to a single system (e.g. the failure of a disk or power supply) the whole system goes down. Having all the software fail at the same time is bad for service availability, but we don't have to worry about how some components can continue operating after others have failed. Partial failures are common in distributed systems:

- one node can crash while others continue running
- occasional network messages may be delayed or lost
- a switch failure may interrupt communication between some nodes, but not others

Distributed systems introduce many new problems that we might never have been forced to address in single systems:

- In a single system it may be very easy to tell that one of the service processes has died (e.g. the process exited with a fatal signal or error return code). In a distributed system our only indication that a component has failed might be that we are no longer receiving messages from it. Perhaps it has failed, or perhaps it is only slow, or perhaps the network link has failed, or perhaps our own network interface has failed. Problems are much more difficult to diagnose in a distributed system, and if we incorrectly diagnose a problem we are likely to choose the wrong solution.
- If we expect a distributed system to continue operating despite the failures of individual components, all of the components need to be made more robust (eg. greater error checking, automatic fail-over, recovery and connection reestablishment). One particularly tricky part of recovery is how to handle situations where a failed component was holding resource locks. We must find some way of recognizing the problem and breaking the locks. And after we have broken the locks we need some way of (a) restoring the resource to a clean state and (b) preventing the previous owner from attempting to continue using the resource if he returns.

### Complexity of Distributed State

Within a single computer system all system resource updates are correctly serialized and we can:

- place all operations on a single time-time (a total ordering)

- at any moment, say what the state of every resource in the system is.

Neither of these is true in a distributed system:

- Distinct nodes in a distributed system operate completely independently of one-another. Unless operations are performed by message exchanges, it is generally not possible to say whether a particular operation on node A happened before or after a different operation on node B. And even when operations are performed via message exchanges, two nodes may disagree on the relative ordering of two events (depending on the order in which each node received the messages).
- Because of the independence of parallel events, different nodes may at any given instant, consider a single resource to be in different states. Thus a resource does not actually have a single state. Rather its state is a vector of the state that the resource is considered to be in by each node in the system.

In single systems, when we needed before-or-after atomicity, we created a single mutex (perhaps in the operating system, or in memory shared by all contending threads). A similar effect can be achieved by sending messages to a central coordinator ... except that those messages are roughly a million times as expensive as operations on an in-memory mutex. This means that serialization approaches that worked very well in a single system can become prohibitively expensive in a distributed system.

## Complexity of Management

In a single computer system has a single configuration. A thousand different systems may each be configured differently:

- they may have different databases of known users
- their services may be configured with different options
- they may have different lists of which servers perform which functions
- their switches may be configured with different routing and fire-wall rules

And even if we create a distributed management service to push management updates out to all nodes:

- some nodes may not be up when the updates are sent, and so not learn of them
- networking problems may create isolated islands of nodes that are operating with a different configuration

## Much Higher Loads

One of the reasons we build distributed systems is to handle increasing loads. Higher loads often uncover weaknesses that had never caused problems under lighter loads. When a load increases by more than a power of ten, it is common to discover new bottlenecks. More nodes mean more messages, which may result in increased overhead, and longer delays. Increased overhead may result in poor scaling, or even in performance that drops as the system size grows. Longer (and more variable) delays often turn up race-conditions that had previously been highly unlikely.

## Heterogeneity

In a single computer system, all of the applications:

- are running on the same instruction set architecture
- are running on the same version of the same operating system
- are using the same versions of the same libraries
- directly interact with one-another through the operating system

In a distributed system, each node may be:

- a different instruction set architecture
- running a different operating system
- running different versions of the software and protocols

and the components interact with one-another through a variety of different networks and file systems. The combinatorics and constant evolution of possible component versions and interconnects render exhaustive testing impossible. These challenges often give rise to interoperability problems and unfortunate interactions that would never happen in a single (homogeneous) system.

## Emergent phenomena

The human mind renders complex systems understandable by contructing simpler abstract models. But simple models (almost by definition) cannot fully capture the behavior of a complex system. Complex systems often exhibit *emergent behaviors* that were not present in the constituent components, but arise from their interactions at scale (e.g. delay-induced oscillations in under-damped feed-back loops). If these phenomena do not happen in smaller systems, we can only learn about them through (hard) experience.

# 47

# Distributed Systems

Distributed systems have changed the face of the world. When your web browser connects to a web server somewhere else on the planet, it is participating in what seems to be a simple form of a **client/server** distributed system. When you contact a modern web service such as Google or Facebook, you are not just interacting with a single machine, however; behind the scenes, these complex services are built from a large collection (i.e., thousands) of machines, each of which cooperate to provide the particular service of the site. Thus, it should be clear what makes studying distributed systems interesting. Indeed, it is worthy of an entire class; here, we just introduce a few of the major topics.

A number of new challenges arise when building a distributed system. The major one we focus on is **failure**; machines, disks, networks, and software all fail from time to time, as we do not (and likely, will never) know how to build "perfect" components and systems. However, when we build a modern web service, we'd like it to appear to clients as if it never fails; how can we accomplish this task?

THE CRUX:
HOW TO BUILD SYSTEMS THAT WORK WHEN COMPONENTS FAIL
How can we build a working system out of parts that don't work correctly all the time? The basic question should remind you of some of the topics we discussed in RAID storage arrays; however, the problems here tend to be more complex, as are the solutions.

Interestingly, while failure is a central challenge in constructing distributed systems, it also represents an opportunity. Yes, machines fail; but the mere fact that a machine fails does not imply the entire system must fail. By collecting together a set of machines, we can build a system that appears to rarely fail, despite the fact that its components fail regularly. This reality is the central beauty and value of distributed systems, and why they underly virtually every modern web service you use, including Google, Facebook, etc.

TIP: COMMUNICATION IS INHERENTLY UNRELIABLE

In virtually all circumstances, it is good to view communication as a fundamentally unreliable activity. Bit corruption, down or non-working links and machines, and lack of buffer space for incoming packets all lead to the same result: packets sometimes do not reach their destination. To build reliable services atop such unreliable networks, we must consider techniques that can cope with packet loss.

Other important issues exist as well. System **performance** is often critical; with a network connecting our distributed system together, system designers must often think carefully about how to accomplish their given tasks, trying to reduce the number of messages sent and further make communication as efficient (low latency, high bandwidth) as possible.

Finally, **security** is also a necessary consideration. When connecting to a remote site, having some assurance that the remote party is who they say they are becomes a central problem. Further, ensuring that third parties cannot monitor or alter an on-going communication between two others is also a challenge.

In this introduction, we'll cover the most basic new aspect that is new in a distributed system: **communication**. Namely, how should machines within a distributed system communicate with one another? We'll start with the most basic primitives available, messages, and build a few higher-level primitives on top of them. As we said above, failure will be a central focus: how should communication layers handle failures?

## 47.1 Communication Basics

The central tenet of modern networking is that communication is fundamentally unreliable. Whether in the wide-area Internet, or a local-area high-speed network such as Infiniband, packets are regularly lost, corrupted, or otherwise do not reach their destination.

There are a multitude of causes for packet loss or corruption. Sometimes, during transmission, some bits get flipped due to electrical or other similar problems. Sometimes, an element in the system, such as a network link or packet router or even the remote host, are somehow damaged or otherwise not working correctly; network cables do accidentally get severed, at least sometimes.

More fundamental however is packet loss due to lack of buffering within a network switch, router, or endpoint. Specifically, even if we could guarantee that all links worked correctly, and that all the components in the system (switches, routers, end hosts) were up and running as expected, loss is still possible, for the following reason. Imagine a packet arrives at a router; for the packet to be processed, it must be placed in memory somewhere within the router. If many such packets arrive at

```
// client code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(20000);
    struct sockaddr_in addrSnd, addrRcv;
    int rc = UDP_FillSockAddr(&addrSnd, "machine.cs.wisc.edu", 10000);
    char message[BUFFER_SIZE];
    sprintf(message, "hello world");
    rc = UDP_Write(sd, &addrSnd, message, BUFFER_SIZE);
    if (rc > 0) {
        int rc = UDP_Read(sd, &addrRcv, message, BUFFER_SIZE);
    }
    return 0;
}

// server code
int main(int argc, char *argv[]) {
    int sd = UDP_Open(10000);
    assert(sd > -1);
    while (1) {
        struct sockaddr_in addr;
        char message[BUFFER_SIZE];
        int rc = UDP_Read(sd, &addr, message, BUFFER_SIZE);
        if (rc > 0) {
            char reply[BUFFER_SIZE];
            sprintf(reply, "goodbye world");
            rc = UDP_Write(sd, &addr, reply, BUFFER_SIZE);
        }
    }
    return 0;
}
```

Figure 47.1: **Example UDP/IP Client/Server Code**

once, it is possible that the memory within the router cannot accommodate all of the packets. The only choice the router has at that point is to **drop** one or more of the packets. This same behavior occurs at end hosts as well; when you send a large number of messages to a single machine, the machine's resources can easily become overwhelmed, and thus packet loss again arises.

Thus, packet loss is fundamental in networking. The question thus becomes: how should we deal with it?

## 47.2 Unreliable Communication Layers

One simple way is this: we don't deal with it. Because some applications know how to deal with packet loss, it is sometimes useful to let them communicate with a basic unreliable messaging layer, an example of the **end-to-end argument** one often hears about (see the **Aside** at end of chapter). One excellent example of such an unreliable layer is found in the **UDP/IP** networking stack available today on virtually all modern systems. To use UDP, a process uses the **sockets** API in order to create a **communication endpoint**; processes on other machines (or on the same machine) send UDP **datagrams** to the original process (a datagram is a fixed-sized message up to some max size).

```
int UDP_Open(int port) {
    int sd;
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) { return -1; }
    struct sockaddr_in myaddr;
    bzero(&myaddr, sizeof(myaddr));
    myaddr.sin_family      = AF_INET;
    myaddr.sin_port        = htons(port);
    myaddr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr *) &myaddr, sizeof(myaddr)) == -1) {
        close(sd);
        return -1;
    }
    return sd;
}

int UDP_FillSockAddr(struct sockaddr_in *addr, char *hostName, int port) {
    bzero(addr, sizeof(struct sockaddr_in));
    addr->sin_family = AF_INET;          // host byte order
    addr->sin_port   = htons(port);      // short, network byte order
    struct in_addr *inAddr;
    struct hostent *hostEntry;
    if ((hostEntry = gethostbyname(hostName)) == NULL) { return -1; }
    inAddr = (struct in_addr *) hostEntry->h_addr;
    addr->sin_addr = *inAddr;
    return 0;
}

int UDP_Write(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int addrLen = sizeof(struct sockaddr_in);
    return sendto(sd, buffer, n, 0, (struct sockaddr *) addr, addrLen);
}

int UDP_Read(int sd, struct sockaddr_in *addr, char *buffer, int n) {
    int len = sizeof(struct sockaddr_in);
    return recvfrom(sd, buffer, n, 0, (struct sockaddr *) addr,
                    (socklen_t *) &len);
}
```

Figure 47.2: **A Simple UDP Library**

Figures 47.1 and 47.2 show a simple client and server built on top of UDP/IP. The client can send a message to the server, which then responds with a reply. With this small amount of code, you have all you need to begin building distributed systems!

UDP is a great example of an unreliable communication layer. If you use it, you will encounter situations where packets get lost (dropped) and thus do not reach their destination; the sender is never thus informed of the loss. However, that does not mean that UDP does not guard against any failures at all. For example, UDP includes a **checksum** to detect some forms of packet corruption.

However, because many applications simply want to send data to a destination and not worry about packet loss, we need more. Specifically, we need reliable communication on top of an unreliable network.

TIP: USE CHECKSUMS FOR INTEGRITY

Checksums are a commonly-used method to detect corruption quickly and effectively in modern systems. A simple checksum is addition: just sum up the bytes of a chunk of data; of course, many other more sophisticated checksums have been created, including basic cyclic redundancy codes (CRCs), the Fletcher checksum, and many others [MK09].

In networking, checksums are used as follows. Before sending a message from one machine to another, compute a checksum over the bytes of the message. Then send both the message and the checksum to the destination. At the destination, the receiver computes a checksum over the incoming message as well; if this computed checksum matches the sent checksum, the receiver can feel some assurance that the data likely did not get corrupted during transmission.

Checksums can be evaluated along a number of different axes. Effectiveness is one primary consideration: does a change in the data lead to a change in the checksum? The stronger the checksum, the harder it is for changes in the data to go unnoticed. Performance is the other important criterion: how costly is the checksum to compute? Unfortunately, effectiveness and performance are often at odds, meaning that checksums of high quality are often expensive to compute. Life, again, isn't perfect.

## 47.3 Reliable Communication Layers

To build a reliable communication layer, we need some new mechanisms and techniques to handle packet loss. Let us consider a simple example in which a client is sending a message to a server over an unreliable connection. The first question we must answer: how does the sender know that the receiver has actually received the message?

The technique that we will use is known as an **acknowledgment**, or **ack** for short. The idea is simple: the sender sends a message to the receiver; the receiver then sends a short message back to *acknowledge* its receipt. Figure 47.3 depicts the process.

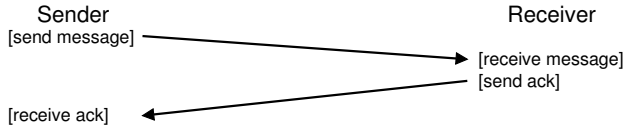


Figure 47.3: **Message Plus Acknowledgment**

When the sender receives an acknowledgment of the message, it can then rest assured that the receiver did indeed receive the original message. However, what should the sender do if it does not receive an acknowledgment?

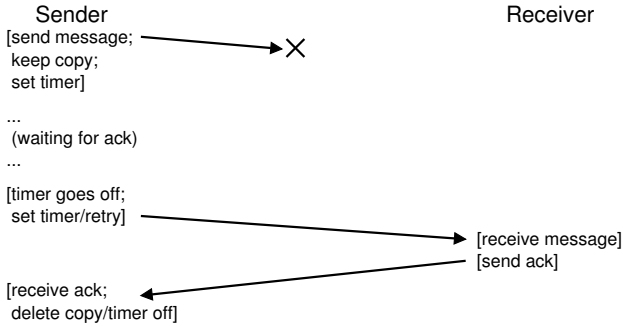


Figure 47.4: **Message Plus Acknowledgment: Dropped Request**

To handle this case, we need an additional mechanism, known as a **timeout**. When the sender sends a message, the sender now sets a timer to go off after some period of time. If, in that time, no acknowledgment has been received, the sender concludes that the message has been lost. The sender then simply performs a **retry** of the send, sending the same message again with hopes that this time, it will get through. For this approach to work, the sender must keep a copy of the message around, in case it needs to send it again. The combination of the timeout and the retry have led some to call the approach **timeout/retry**; pretty clever crowd, those networking types, no? Figure 47.4 shows an example.

Unfortunately, timeout/retry in this form is not quite enough. Figure 47.5 shows an example of packet loss which could lead to trouble. In this example, it is not the original message that gets lost, but the acknowledgment. From the perspective of the sender, the situation seems the same: no ack was received, and thus a timeout and retry are in order. But from the perspective of the receiver, it is quite different: now the same message has been received twice! While there may be cases where this is OK, in general it is not; imagine what would happen when you are downloading a file and extra packets are repeated inside the download. Thus, when we are aiming for a reliable message layer, we also usually want to guarantee that each message is received **exactly once** by the receiver.

To enable the receiver to detect duplicate message transmission, the sender has to identify each message in some unique way, and the receiver needs some way to track whether it has already seen each message before. When the receiver sees a duplicate transmission, it simply acks the message, but (critically) does *not* pass the message to the application that receives the data. Thus, the sender receives the ack but the message is not received twice, preserving the exactly-once semantics mentioned above.

There are myriad ways to detect duplicate messages. For example, the sender could generate a unique ID for each message; the receiver could track every ID it has ever seen. This approach could work, but it is prohibitively costly, requiring unbounded memory to track all IDs.

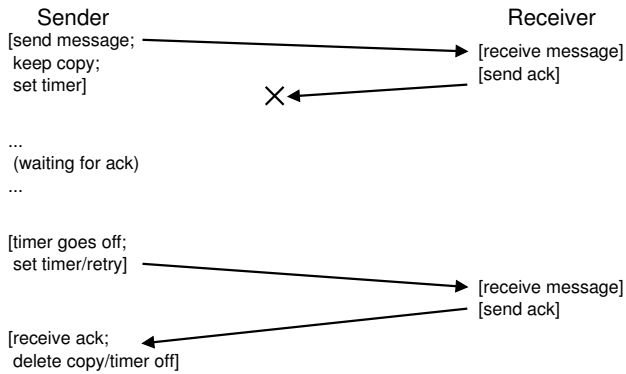


Figure 47.5: **Message Plus Acknowledgment: Dropped Reply**

A simpler approach, requiring little memory, solves this problem, and the mechanism is known as a **sequence counter**. With a sequence counter, the sender and receiver agree upon a start value (e.g., 1) for a counter that each side will maintain. Whenever a message is sent, the current value of the counter is sent along with the message; this counter value ($N$) serves as an ID for the message. After the message is sent, the sender then increments the value (to $N + 1$).

The receiver uses its counter value as the expected value for the ID of the incoming message from that sender. If the ID of a received message ($N$) matches the receiver's counter (also $N$), it acks the message and passes it up to the application; in this case, the receiver concludes this is the first time this message has been received. The receiver then increments its counter (to $N + 1$), and waits for the next message.

If the ack is lost, the sender will timeout and re-send message $N$. This time, the receiver's counter is higher ($N+1$), and thus the receiver knows it has already received this message. Thus it acks the message but does *not* pass it up to the application. In this simple manner, sequence counters can be used to avoid duplicates.

The most commonly used reliable communication layer is known as **TCP/IP**, or just **TCP** for short. TCP has a great deal more sophistication than we describe above, including machinery to handle congestion in the network [VJ88], multiple outstanding requests, and hundreds of other small tweaks and optimizations. Read more about it if you're curious; better yet, take a networking course and learn that material well.

## 47.4 Communication Abstractions

Given a basic messaging layer, we now approach the next question in this chapter: what abstraction of communication should we use when building a distributed system?

TIP: BE CAREFUL SETTING THE TIMEOUT VALUE

As you can probably guess from the discussion, setting the timeout value correctly is an important aspect of using timeouts to retry message sends. If the timeout is too small, the sender will re-send messages needlessly, thus wasting CPU time on the sender and network resources. If the timeout is too large, the sender waits too long to re-send and thus perceived performance at the sender is reduced. The "right" value, from the perspective of a single client and server, is thus to wait just long enough to detect packet loss but no longer.

However, there are often more than just a single client and server in a distributed system, as we will see in future chapters. In a scenario with many clients sending to a single server, packet loss at the server may be an indicator that the server is overloaded. If true, clients might retry in a different adaptive manner; for example, after the first timeout, a client might increase its timeout value to a higher amount, perhaps twice as high as the original value. Such an **exponential back-off** scheme, pioneered in the early Aloha network and adopted in early Ethernet [A70], avoids situations where resources are being overloaded by an excess of re-sends. Robust systems strive to avoid overload of this nature.

The systems community developed a number of approaches over the years. One body of work took OS abstractions and extended them to operate in a distributed environment. For example, **distributed shared memory** (**DSM**) systems enable processes on different machines to share a large, virtual address space [LH89]. This abstraction turns a distributed computation into something that looks like a multi-threaded application; the only difference is that these threads run on different machines instead of different processors within the same machine.

The way most DSM systems work is through the virtual memory system of the OS. When a page is accessed on one machine, two things can happen. In the first (best) case, the page is already local on the machine, and thus the data is fetched quickly. In the second case, the page is currently on some other machine. A page fault occurs, and the page fault handler sends a message to some other machine to fetch the page, install it in the page table of the requesting process, and continue execution.

This approach is not widely in use today for a number of reasons. The largest problem for DSM is how it handles failure. Imagine, for example, if a machine fails; what happens to the pages on that machine? What if the data structures of the distributed computation are spread across the entire address space? In this case, parts of these data structures would suddenly become unavailable. Dealing with failure when parts of your address space go missing is hard; imagine a linked list that where a next pointer points into a portion of the address space that is gone. Yikes!

A further problem is performance. One usually assumes, when writing code, that access to memory is cheap. In DSM systems, some accesses

are inexpensive, but others cause page faults and expensive fetches from remote machines. Thus, programmers of such DSM systems had to be very careful to organize computations such that almost no communication occurred at all, defeating much of the point of such an approach. Though much research was performed in this space, there was little practical impact; nobody builds reliable distributed systems using DSM today.

## 47.5 Remote Procedure Call (RPC)

While OS abstractions turned out to be a poor choice for building distributed systems, programming language (PL) abstractions make much more sense. The most dominant abstraction is based on the idea of a **remote procedure call**, or **RPC** for short [BN84][1].

Remote procedure call packages all have a simple goal: to make the process of executing code on a remote machine as simple and straightforward as calling a local function. Thus, to a client, a procedure call is made, and some time later, the results are returned. The server simply defines some routines that it wishes to export. The rest of the magic is handled by the RPC system, which in general has two pieces: a **stub generator** (sometimes called a **protocol compiler**), and the **run-time library**. We'll now take a look at each of these pieces in more detail.

### Stub Generator

The stub generator's job is simple: to remove some of the pain of packing function arguments and results into messages by automating it. Numerous benefits arise: one avoids, by design, the simple mistakes that occur in writing such code by hand; further, a stub compiler can perhaps optimize such code and thus improve performance.

The input to such a compiler is simply the set of calls a server wishes to export to clients. Conceptually, it could be something as simple as this:

```
interface {
  int func1(int arg1);
  int func2(int arg1, int arg2);
};
```

The stub generator takes an interface like this and generates a few different pieces of code. For the client, a **client stub** is generated, which contains each of the functions specified in the interface; a client program wishing to use this RPC service would link with this client stub and call into it in order to make RPCs.

Internally, each of these functions in the client stub do all of the work needed to perform the remote procedure call. To the client, the code just

[1] In modern programming languages, we might instead say **remote method invocation** (**RMI**), but who likes these languages anyhow, with all of their fancy objects?

appears as a function call (e.g., the client calls `func1(x)`); internally, the code in the client stub for `func1()` does this:

- **Create a message buffer.** A message buffer is usually just a contiguous array of bytes of some size.
- **Pack the needed information into the message buffer.** This information includes some kind of identifier for the function to be called, as well as all of the arguments that the function needs (e.g., in our example above, one integer for `func1`). The process of putting all of this information into a single contiguous buffer is sometimes referred to as the **marshaling** of arguments or the **serialization** of the message.
- **Send the message to the destination RPC server.** The communication with the RPC server, and all of the details required to make it operate correctly, are handled by the RPC run-time library, described further below.
- **Wait for the reply.** Because function calls are usually **synchronous**, the call will wait for its completion.
- **Unpack return code and other arguments.** If the function just returns a single return code, this process is straightforward; however, more complex functions might return more complex results (e.g., a list), and thus the stub might need to unpack those as well. This step is also known as **unmarshaling** or **deserialization**.
- **Return to the caller.** Finally, just return from the client stub back into the client code.

For the server, code is also generated. The steps taken on the server are as follows:

- **Unpack the message.** This step, called **unmarshaling** or **deserialization**, takes the information out of the incoming message. The function identifier and arguments are extracted.
- **Call into the actual function.** Finally! We have reached the point where the remote function is actually executed. The RPC runtime calls into the function specified by the ID and passes in the desired arguments.
- **Package the results.** The return argument(s) are marshaled back into a single reply buffer.
- **Send the reply.** The reply is finally sent to the caller.

There are a few other important issues to consider in a stub compiler. The first is complex arguments, i.e., how does one package and send a complex data structure? For example, when one calls the `write()` system call, one passes in three arguments: an integer file descriptor, a pointer to a buffer, and a size indicating how many bytes (starting at the pointer) are to be written. If an RPC package is passed a pointer, it needs to be able to figure out how to interpret that pointer, and perform the

correct action. Usually this is accomplished through either well-known types (e.g., a `buffer_t` that is used to pass chunks of data given a size, which the RPC compiler understands), or by annotating the data structures with more information, enabling the compiler to know which bytes need to be serialized.

Another important issue is the organization of the server with regards to concurrency. A simple server just waits for requests in a simple loop, and handles each request one at a time. However, as you might have guessed, this can be grossly inefficient; if one RPC call blocks (e.g., on I/O), server resources are wasted. Thus, most servers are constructed in some sort of concurrent fashion. A common organization is a **thread pool**. In this organization, a finite set of threads are created when the server starts; when a message arrives, it is dispatched to one of these worker threads, which then does the work of the RPC call, eventually replying; during this time, a main thread keeps receiving other requests, and perhaps dispatching them to other workers. Such an organization enables concurrent execution within the server, thus increasing its utilization; the standard costs arise as well, mostly in programming complexity, as the RPC calls may now need to use locks and other synchronization primitives in order to ensure their correct operation.

### Run-Time Library

The run-time library handles much of the heavy lifting in an RPC system; most performance and reliability issues are handled herein. We'll now discuss some of the major challenges in building such a run-time layer.

One of the first challenges we must overcome is how to locate a remote service. This problem, of **naming**, is a common one in distributed systems, and in some sense goes beyond the scope of our current discussion. The simplest of approaches build on existing naming systems, e.g., hostnames and port numbers provided by current internet protocols. In such a system, the client must know the hostname or IP address of the machine running the desired RPC service, as well as the port number it is using (a port number is just a way of identifying a particular communication activity taking place on a machine, allowing multiple communication channels at once). The protocol suite must then provide a mechanism to route packets to a particular address from any other machine in the system. For a good discussion of naming, you'll have to look elsewhere, e.g., read about DNS and name resolution on the Internet, or better yet just read the excellent chapter in Saltzer and Kaashoek's book [SK09].

Once a client knows which server it should talk to for a particular remote service, the next question is which transport-level protocol should RPC be built upon. Specifically, should the RPC system use a reliable protocol such as TCP/IP, or be built upon an unreliable communication layer such as UDP/IP?

Naively the choice would seem easy: clearly we would like for a request to be reliably delivered to the remote server, and clearly we would

like to reliably receive a reply. Thus we should choose the reliable transport protocol such as TCP, right?

Unfortunately, building RPC on top of a reliable communication layer can lead to a major inefficiency in performance. Recall from the discussion above how reliable communication layers work: with acknowledgments plus timeout/retry. Thus, when the client sends an RPC request to the server, the server responds with an acknowledgment so that the caller knows the request was received. Similarly, when the server sends the reply to the client, the client acks it so that the server knows it was received. By building a request/response protocol (such as RPC) on top of a reliable communication layer, two "extra" messages are sent.

For this reason, many RPC packages are built on top of unreliable communication layers, such as UDP. Doing so enables a more efficient RPC layer, but does add the responsibility of providing reliability to the RPC system. The RPC layer achieves the desired level of responsibility by using timeout/retry and acknowledgments much like we described above. By using some form of sequence numbering, the communication layer can guarantee that each RPC takes place exactly once (in the case of no failure), or at most once (in the case where failure arises).

### Other Issues

There are some other issues an RPC run-time must handle as well. For example, what happens when a remote call takes a long time to complete? Given our timeout machinery, a long-running remote call might appear as a failure to a client, thus triggering a retry, and thus the need for some care here. One solution is to use an explicit acknowledgment (from the receiver to sender) when the reply isn't immediately generated; this lets the client know the server received the request. Then, after some time has passed, the client can periodically ask whether the server is still working on the request; if the server keeps saying "yes", the client should be happy and continue to wait (after all, sometimes a procedure call can take a long time to finish executing).

The run-time must also handle procedure calls with large arguments, larger than what can fit into a single packet. Some lower-level network protocols provide such sender-side **fragmentation** (of larger packets into a set of smaller ones) and receiver-side **reassembly** (of smaller parts into one larger logical whole); if not, the RPC run-time may have to implement such functionality itself. See Birrell and Nelson's excellent RPC paper for details [BN84].

One issue that many systems handle is that of **byte ordering**. As you may know, some machines store values in what is known as **big endian** ordering, whereas others use **little endian** ordering. Big endian stores bytes (say, of an integer) from most significant to least significant bits, much like Arabic numerals; little endian does the opposite. Both are equally valid ways of storing numeric information; the question here is how to communicate between machines of *different* endianness.

Aside: **The End-to-End Argument**

The **end-to-end argument** makes the case that the highest level in a system, i.e., usually the application at "the end", is ultimately the only locale within a layered system where certain functionality can truly be implemented. In their landmark paper [SRC84], Saltzer et al. argue this through an excellent example: reliable file transfer between two machines. If you want to transfer a file from machine $A$ to machine $B$, and make sure that the bytes that end up on $B$ are exactly the same as those that began on $A$, you must have an "end-to-end" check of this; lower-level reliable machinery, e.g., in the network or disk, provides no such guarantee.

The contrast is an approach which tries to solve the reliable-file-transfer problem by adding reliability to lower layers of the system. For example, say we build a reliable communication protocol and use it to build our reliable file transfer. The communication protocol guarantees that every byte sent by a sender will be received in order by the receiver, say using timeout/retry, acknowledgments, and sequence numbers. Unfortunately, using such a protocol does not a reliable file transfer make; imagine the bytes getting corrupted in sender memory before the communication even takes place, or something bad happening when the receiver writes the data to disk. In those cases, even though the bytes were delivered reliably across the network, our file transfer was ultimately not reliable. To build a reliable file transfer, one must include end-to-end checks of reliability, e.g., after the entire transfer is complete, read back the file on the receiver disk, compute a checksum, and compare that checksum to that of the file on the sender.

The corollary to this maxim is that sometimes having lower layers provide extra functionality can indeed improve system performance or otherwise optimize a system. Thus, you should not rule out having such machinery at a lower-level in a system; rather, you should carefully consider the utility of such machinery, given its eventual usage in an overall system or application.

RPC packages often handle this by providing a well-defined endianness within their message formats. In Sun's RPC package, the **XDR** (**eXternal Data Representation**) layer provides this functionality. If the machine sending or receiving a message matches the endianness of XDR, messages are just sent and received as expected. If, however, the machine communicating has a different endianness, each piece of information in the message must be converted. Thus, the difference in endianness can have a small performance cost.

A final issue is whether to expose the asynchronous nature of communication to clients, thus enabling some performance optimizations. Specifically, typical RPCs are made **synchronously**, i.e., when a client issues the procedure call, it must wait for the procedure call to return

before continuing. Because this wait can be long, and because the client may have other work it could be doing, some RPC packages enable you to invoke an RPC **asynchronously**. When an asynchronous RPC is issued, the RPC package sends the request and returns immediately; the client is then free to do other work, such as call other RPCs or other useful computation. The client at some point will want to see the results of the asynchronous RPC; it thus calls back into the RPC layer, telling it to wait for outstanding RPCs to complete, at which point return arguments can be accessed.

## 47.6 Summary

We have seen the introduction of a new topic, distributed systems, and its major issue: how to handle failure which is now a commonplace event. As they say inside of Google, when you have just your desktop machine, failure is rare; when you're in a data center with thousands of machines, failure is happening all the time. The key to any distributed system is how you deal with that failure.

We have also seen that communication forms the heart of any distributed system. A common abstraction of that communication is found in remote procedure call (RPC), which enables clients to make remote calls on servers; the RPC package handles all of the gory details, including timeout/retry and acknowledgment, in order to deliver a service that closely mirrors a local procedure call.

The best way to really understand an RPC package is of course to use one yourself. Sun's RPC system, using the stub compiler `rpcgen`, is a common one, and is widely available on systems today, including Linux. Try it out, and see what all the fuss is about.

## References

[A70] "The ALOHA System — Another Alternative for Computer Communications"
Norman Abramson
The 1970 Fall Joint Computer Conference
*The ALOHA network pioneered some basic concepts in networking, including exponential back-off and retransmit, which formed the basis for communication in shared-bus Ethernet networks for years.*

[BN84] "Implementing Remote Procedure Calls"
Andrew D. Birrell, Bruce Jay Nelson
ACM TOCS, Volume 2:1, February 1984
*The foundational RPC system upon which all others build. Yes, another pioneering effort from our friends at Xerox PARC.*

[MK09] "The Effectiveness of Checksums for Embedded Control Networks"
Theresa C. Maxino and Philip J. Koopman
IEEE Transactions on Dependable and Secure Computing, 6:1, January '09
*A nice overview of basic checksum machinery and some performance and robustness comparisons between them.*

[LH89] "Memory Coherence in Shared Virtual Memory Systems"
Kai Li and Paul Hudak
ACM TOCS, 7:4, November 1989
*The introduction of software-based shared memory via virtual memory. An intriguing idea for sure, but not a lasting or good one in the end.*

[SK09] "Principles of Computer System Design"
Jerome H. Saltzer and M. Frans Kaashoek
Morgan-Kaufmann, 2009
*An excellent book on systems, and a must for every bookshelf. One of the few terrific discussions on naming we've seen.*

[SRC84] "End-To-End Arguments in System Design"
Jerome H. Saltzer, David P. Reed, David D. Clark
ACM TOCS, 2:4, November 1984
*A beautiful discussion of layering, abstraction, and where functionality must ultimately reside in computer systems.*

[VJ88] "Congestion Avoidance and Control"
Van Jacobson
SIGCOMM '88
*A pioneering paper on how clients should adjust to perceived network congestion; definitely one of the key pieces of technology underlying the Internet, and a must read for anyone serious about systems, and for Van Jacobson's relatives because well relatives should read all of your papers.*

# Distributed Systems

Monday, November 21, 2016 6:12 PM

(Representational state transfer) RESTful interfaces

- https://www.wikiwand.com/en/Representational_state_transfer
- **Representational state transfer** (**REST**) or **RESTful** web services are one way of providing interoperability between computer systems on the Internet. REST-compliant web services allow requesting systems to access and manipulate textual representations of web resources using a uniform and predefined set of stateless operations. Other forms of web service exist, which expose their own arbitrary sets of operations such as WSDL and SOAP.[1] "Web resources" were first defined on the World Wide Web as documents or files identified by their URLs, but today they have a much more generic and abstract definition encompassing every thing or entity that can be identified, named, addressed or handled, in any way whatsoever, on the web. In a REST web service, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSONor some other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide hypertext links to other related resources or collections of resources. Using HTTP, as is most common, the kind of operations available include those predefined by the HTTP verbs GET, POST, PUT, DELETE and so on. By making use of a stateless protocol and standard operations REST systems aim for fast performance, reliability, and the ability to grow, by using reused components that can be managed and updated without affecting the system as a whole, even while it is running
- Architectural constraints
  - Client-Server
    - Separation allows components to evolve independently supporting Internet-scale requirement
    - Improves portability of user interface across multiple platforms
    - Improves scalability by simplifying the server

components

- Stateless
  - No client context stored on the server between requests
  - The session state is held in the client and can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication
  - Client begins sending requests when it is ready to make the transition to a new state
    - When requests are outstanding, client is considered to be in transition.
- Cacheable
  - Responses must be defined as cacheable or not to prevent clients from reusing stale or inappropriate data in response to further requests
  - Well-managed caching partially/completely eliminates some client-server interactions, improving scalability and performance
- Layered System
  - Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches
    - May also enforce security policies
- Code on demand (optional)
  - Servers can temporarily extend or customize the functionality of a client by the transfer of executable code
    - Compiled components such as Java applets, javascript scripts
- Uniform Interface
  - Simplifies and decouples architecture, enabling each part to evolve independently
  - <u>Constraints</u>
    - Identification of resources
      - Resources are conceptually separate from the representations that are returned to the client
      - Server may send data from its database as HTML, XML, JSON, but may not be the server's internal representation
    - Manipulation of resources through representations
      - When client holds a representation of a

resource, including any metadata attached, it has enough information to modify or delete the resource.

- Self-descriptive messages
  - Each message includes enough informatino to describe how to process the message
- Hypermedia as the engine of application state
  - REST client should be able to use server-provided links dynamically to discover all available actions and resources it needs
  - As access proceeds the server responds with text that include hyperlinks to other actions that are currently available
  - No need for the client to be hard coded with information regarding the structure or dynamics of the REST service

Lease

- https://www.wikiwand.com/en/Lease_(computer_science)
- In computer science, a **Lease** is a contract that gives its holder specified rights to some resource for a limited period. Because it is time-limited, a lease is an alternative to a lock for resource serialization.
- Motivation
  - A traditional resource lock is granted until it is explicitly released by the locking client process. Reasons why a lock might not be released include:
    - The client failed before releasing the resources
    - The client deadlocked while attempting to allocate another resource
    - The client was blocked or delayed for an unreasonable period
    - The client neglected to free the resource, perhaps due to a bug
    - The request to free the resource was lost
    - The resource manager failed or lost track of the resource stated
  - Any of these could end the availability of an important reusable resource until the system is reset. By contract, a lease is valid for a limited period, after which it automatically expires, making the

resource available for reallocation by a new client.

- Leases are commonly used in distributed systems for applications ranging from DHCP address allocation to file locking, but they are not (by themselves) a complete solution:
- Problems
  - There must be some means of notifying the lease holder of the expiration and preventing that agent from continuing to rely on the resource. Often, this is done by requiring all requests to be accompanied by an access token, which is invalidated if the associated lease has expired.
  - If a lease is revoked after the lease holder has started operating on the resource, revocation may leave the resource in a compromised state. In such situations, it is common to use Atomic transactions to ensure that updates that do not complete have no effect.

Consensus

- https://www.wikiwand.com/en/Consensus_(computer_science)
- A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires processes to agree on some data value that is needed during computation.
  - Examples of applications of **consensus** include
    - whether to commit a transaction to a database,
    - agreeing on the identity of a leader,
    - state machine replication,
    - atomic broadcasts.
- One approach to generating consensus is for all processes (agents) to agree on a majority value. In this context, a majority requires at least one more than half of available votes (where each process is given a vote).
  - However one or more faulty processes may skew the resultant outcome such that consensus may not be reached or reached incorrectly.
- Protocols that solve consensus problems are designed to deal with limited numbers of faulty processes
  - Protocols must satisfy a number of requirements messing with input and output values.
- Properties of protocol
  - Termination
    - Every correct process decides some value
  - Validity

- If all processes propose the same value v then all correct processes decide v

- Integrity
  - Every correct process decides at most one value, and if it decides some value v then v must have been proposed by some process
- Agreement
  - Every correct process must agree on the same value

# Distributed System Security

## Introduction

An operating system can only control its own machine's resources. Thus, operating systems will have challenges in providing security in distributed systems, where more than one machine must cooperate. There are two large problems:

1. The other machines in the distributed system might not properly implement the security policies you want.
2. Machines in a distributed system communicate across a network that none of them fully control and that, generally, cannot be trusted.

As suggested earlier, cryptography will be the major tool we use here, but we also said cryptography was hard to get right. That makes it sound like the perfect place to use carefully designed standard tools, rather than to expect everyone to build their own. That's precisely correct. So,

THE CRUX OF THE PROBLEM
HOW TO PROTECT DISTRIBUTED SYSTEM OPERATIONS?

How can we secure a system spanning more than one machine? What tools are available to help us protect such systems? How do we use them properly? What are the areas in using the tools that require us to be careful and thoughtful?

## The Role of Authentication

How can we handle our uncertainty about whether our partners in a distributed system are going to enforce our security policies? In most cases, we can't do much. At best, we can try to arrange to agree on policies and hope everyone follows through on those agreements. There are some special cases where we can get high-quality evidence that our partners have behaved properly, but that's not easy, in general. For example, how can we know that they are using full disk encryption, or that they have carefully wiped an encryption key we are finished using, or that they have set access controls on the local copies of their files properly? They can say they did, but how can we know?

Generally, we can't. But you're used to that. In the real world, your friends and relatives know some secrets about you, and they might have keys to get into your home, and if you loan them your car you're pretty sure you'll get it back. That's not so much because you have perfect mechanisms to prevent those trusted parties from behaving badly, but because you are pretty sure they won't. If you're wrong, perhaps you can detect that they haven't behaved well and take compensating actions (like changing your locks or calling the police to report your car stolen). We'll need to rely on the same factors in distributed computer systems. We will simply have to trust that some parties will behave well. In some cases, we can detect when they don't and adjust our trust in the parties accordingly, and maybe take other compensating actions.

Of course, in the cyber world, our actions are at a distance over a network, and all we see are bits going out and coming in on the network. For a trust-based solution to work, we have to be quite sure that the bits we send out can be verified by our buddies as truly coming from us, and we have to be sure that the bits coming in really were created by them. That's a job for authentication. As suggested in the earlier authentication chapter, when working over a network, we need to authenticate based on a bundle of bits. Most commonly, we use a form of authentication based on what you know. Now, think back to the earlier chapters. What might someone running on a remote operating system know that no one else knows? How about a password? How about a private key?

Most of our distributed system authentication will rely on one of these two elements. Either you require the remote machine to provide you with a password, or you require it to provide evidence using a private key stored only on that machine[1]. In each case, you need to know something: either the password (or, better, a cryptographic hash of the password plus a salt) or the public key.

When is each appropriate? Passwords tend to be useful if there are a vast number of parties who need to authenticate themselves to one party. Public keys tend to be useful if there's one party who needs to authenticate himself to a vast number of parties. Why? With a password, the authentication provides evidence that somebody knows a password. If you want to know exactly who that is (which is usually important), only the party authenticating and the party checking can know it. With a public key, many parties can know the key, but only one party who knows the matching private key can authenticate himself. So we tend to use both mechanisms, but for different cases. When a web site authenticates itself to a user, it's done with PK cryptography. By distributing one single public key (to vast numbers of users), the web site can be authenticated by all its users. The web site need not bother keeping separate authentication information to authenticate itself to each user. When that user authenticates itself to the web site, it's done with a password. Each user must be separately authenticated to the web site, so we require a unique bit of identifying information for that user, preferably something that's easy for a person to use. Setting up and distributing public keys is hard, while setting up individual passwords is relatively easy.

How, practically, do we use each of these authentication mechanisms in a distributed system? If we want a remote partner to authenticate itself via passwords, we will require it to provide us with that password, which we will check. We'll need to encrypt the transport of the password across the network if we do that, since otherwise anyone eavesdropping on the network (which is easy for many wireless networks) will readily learn passwords sent unencrypted. Encrypting the password will require that we already have either a shared symmetric key or our partner's public key. So let's concentrate for

---

[1] We occasionally use other methods, such as smart cards or remote biometric readers. They are less common in today's systems, though. If you understand how we use passwords and public key cryptography for distributed system authentication, you can probably figure out how to make proper use of these other techniques, too. If you don't, you'll be better off figuring out the common techniques before moving to the less common ones.

the moment on how we handle getting that public key, either to use it directly or to set up the cryptography to protect the password in transit.

We'll spend the rest of the chapter on securing the network connection, but please don't forget that even if you secure the network perfectly, you still face the major security challenge of the uncontrolled site you're interacting with on the other side of the network. If your compromised partner attacks you, it will offer little consolation that his attack was authenticated and encrypted.

## Public Key Authentication for Distributed Systems

The public key doesn't need to be secret, but we need to be sure it really belongs to our partner. If we have a face-to-face meeting with him, he can give us his public key in some form or another, in which case we can be pretty sure it's his. That's limiting, though, since we often interact with partners who we never see face to face. For that matter, whose "face" belongs to Amazon or Google?

Fortunately, we can use the fact that secrecy isn't required to simply create a bunch of bits containing the public key. Anyone who gets a copy of the bits has the key. But how do they know for sure whose key it is? What if some other trusted party known to everyone who needs to authenticate our partner used their own public key to cryptographically sign that bunch of bits, verifying that they do indeed belong to our partner? If we could check that signature, we could then be sure that bunch of bits really does represent his public key, at least to the extent that we trust that third party who did the signature.

This technique is how we actually authenticate web sites and many other entities on the Internet. Every time you browse the web or perform any other web-based activity, you use it. The signed bundle of bits is called a *certificate*. Essentially, it contains information about the party that owns the public key, the public key itself, and other information, such as an expiration date. The entire set of information, including the public key, is run through a cryptographic hash, and the result is encrypted with the trusted third party's private key, digitally signing the certificate. If you obtain a copy of the certificate, and can check the signature, you can learn someone else's public key, even if you have never met or had any direct interaction with them. In certain ways, it's a beautiful technology that empowers the whole Internet.

Let's briefly go through an example, to solidify the concepts. Let's say Frobazz Inc. wants to obtain a certificate for its public key, which is $K_F$. Frobazz Inc. pays big bucks to Acmesign Co., a widely trusted company whose business it is to sell certificates. Such companies are commonly called Certificate Authorities, or CAs, since they create authoritative certificates trusted by many parties. Acmesign checks up on Frobazz Inc. to ensure that the people asking for the certificate actually are legitimate representatives of Frobazz. Acmesign then makes very, very sure that the public key it's about to embed in a certificate actually is the one that Frobazz wants to use. Assuming it is, Acmesign runs a cryptographic hashing algorithm (perhaps SHA-3) on Frobazz's name, public key $K_F$, and other information, producing hash $H_F$. Acmesign then encrypts $H_F$ with its own private key, $P_A$, producing digital signature $S_F$. Finally, Acmesign combines all of the

information used to produce $H_F$, plus Acmesign's own identity and the signature $S_F$, into the certificate $C_F$, which it hands over to Frobazz, presumably in exchange for a bunch of money. Remember, $C_F$ is just a bunch of bits.

Now Frobazz Inc. wants to authenticate itself over the Internet to one of its customers. If the customer already has Frobazz's public key, we can use public key authentication mechanisms directly. If the customer does not have the public key, Frobazz sends $C_F$ to the customer. The customer examines the certificate, sees that it was generated by Acmesign using, say, SHA-3, and runs the same information that Acmesign hashed (all of which is in the certificate itself) through SHA-3, producing $H_F'$. Then the customer uses Acmesign's public key to decrypt $S_F$ (also in the certificate), obtaining $H_F$. If all is well, $H_F$ equals $H_F'$, and now the customer knows that the public key in the certificate is indeed Frobazz's. Public key authentication can proceed[2]. If the two hashes aren't exactly the same, the customer knows that something fishy is going on and will not accept the certificate.

There are some wonderful properties about this approach to learning public keys. First, note that the signing authority (Acmesign, in our example) did not need to participate in the process of the customer checking the certificate. In fact, Frobazz didn't really, either. The customer can get the certificate from literally anywhere and obtain the same degree of assurance of its validity. Second, it only needs to be done once per customer. After obtaining the certificate and checking it, the customer has the public key he needs. From that point onward, he can simply store it and use it. If, for whatever reason, he loses it, he can either extract it again from the certificate (if that has been saved), or go through the process of obtaining the certificate all over again. Third, the customer had no need to trust the party claiming to be Frobazz until that identity had been proven by checking the certificate. The customer can keep that party at arm's length and proceed with caution until the certificate checks out.

Assuming you've been paying attention for the last few chapters, you should be saying to yourself, "now, wait a minute, isn't there a chicken-and-egg problem here?" We'll learn Frobazz's public key by getting a certificate for it. The certificate will be signed by Acmesign. We'll check the signature by knowing Acmesign's public key. But where did we get Acmesign's key? We really hope you did have that head-scratching moment and asked yourself that question, because if you did, you understand the true nature of the Internet authentication problem. Ultimately, we've got to bootstrap it. You've got to somehow or other obtain a public key for somebody that you trust. Once you do, if it's the right public key for the right kind of party, you can then obtain a lot of other public keys. But without something to start from, you can't do much of anything.

Where do you get that primal public key? Most commonly, it comes in a piece of software you obtain and install. The one you use most often is probably your browser, which typically comes with the public keys for several hundred trusted authorities.

---

[2] And, indeed, must, since all this business with checking the certificate merely told the customer what Frobazz's public key was. It did nothing to assure the customer that whoever sent him the certificate actually was Frobazz or knew Frobazz's private key.

Whenever you go to a new web site that cares about security, it provides you with a certificate containing that site's public key, and signed by one of those trusted authorities pre-configured into your browser. You use the pre-configured public key of that authority to verify that the certificate is indeed proper, after which you know the public key of that web site. From that point onward, you can use the web site's public key to authenticate it. There are some serious caveats here, but let's put those aside for the moment.

Anyone can create a certificate, not just those trusted CAs, either by getting one from someone whose business it is to issue certificates or simply by creating one from scratch, following a certificate standard (X.509 is the most commonly used certificate standard [I12]). The necessary requirement is that the party being authenticated and the parties performing the authentication must all trust whoever created the certificate. If they don't trust that party, why would they believe the certificate is correct?

If you are building your own distributed system, you can create your own certificates from a machine you (and other participants in the system) trust and can handle the bootstrapping issue by carefully hand-installing the certificate signing machine's public key wherever it needs to be. There are a number of existing software packages for creating certificates, and, as usual with critical cryptographic software, you're better off using an existing, trusted implementation rather than coding up one of your own. One example you might want to look at is PGP (available in both supported commercial versions and compatible but less supported free versions) [P16], but there are others. If you are working with a fixed number of machines and you can distribute the public key by hand in some reasonable way, you can dispense entirely with certificates. Remember, the only point of a PK certificate is to distribute the public key, so if your public keys are already where they need to be, you don't need certificates.

OK, one way or another you've obtained the public key you need to authenticate some remote machine. Now what? Well, anything they send you encrypted with their private key will only decrypt with their public key, so anything that decrypts properly with the public key must have come from them, right? Yes, it must have come from them at some point, but it's possible for an adversary to have made a copy of a legitimate message the site sent at some point in the past and then replay it at some future date. Depending on exactly what's going on, that could cause trouble, since you may take actions based on that message that the legitimate site did not ask for. So usually we take measures to ensure that we're not being subjected to a *replay* attack. Such measures generally involve ensuring that each encrypted message contains unique information not in any other message. This feature is built in properly to standard cryptographic protocols, so if you follow our advice and use one of those, you will get protection from such replay attacks. If you insist on building your own cryptography, you'll need to learn a good deal more about this issue and will have to apply that knowledge very carefully. Also, public key cryptography is expensive. We want to stop using it as soon as possible, but we also want to continue to get authentication guarantees. We'll see how to do that when we discuss SSL and TLS.

## Password Authentication for Distributed Systems

The other common option used to authenticate in distributed systems is to use a password. As noted above, that will work best in situations where only two parties need to deal with any particular password: the party being authenticated and the authenticating party. They make sense when an individual user is authenticating himself to a site that hosts many users, such as when you log in to Amazon. They don't make sense when that site is trying to authenticate itself to an individual user, such as when a web site claiming to be Amazon wants to do business with you. Public key authentication works better there.

How do we properly handle password authentication over the network, when it is a reasonable choice? The password is usually associated with a particular user ID, so the user provides that ID and password to the site requiring authentication. That typically happens over a network, and typically we cannot guarantee that networks provide confidentiality. If our password is divulged to someone else, they'll be able to pose as us, so we must add confidentiality to this cross-network authentication, generally by encrypting at least the password itself (though encrypting everything involved is better). So a typical interchange with Alice trying to authenticate herself to Frobazz Inc.'s web site would involve the site requesting a user ID and password and Alice providing both, but encrypting them before sending them over the network.

The obvious question you should ask is, encrypting them with what key? Well, if Frobazz authenticated itself to Alice using PK, as discussed above, Alice can encrypt her user ID and password with Frobazz's public key. Frobazz Inc., having the matching private key, will be able to check them, but nobody else can read them. In actuality, there are various reasons why this alone would not suffice, including replay attacks, as mentioned above. But we can and do use Frobazz's private key to set up cryptography that will protect Alice's password in transit. We'll discuss the details in the section on SSL/TLS.

We discussed issues of password choice and management in the chapter on authentication, and those all apply in the networking context. Otherwise, there's not that much more to say about how we'll use passwords, other than to note that after the remote site has verified the password, what does it actually know? That the site or user who sent the password knows it, and, to the strength of the password, that site or user is who it claims to be. But what about future messages that come in, supposedly from that site? Remember, anyone can create any message they want, so if all we do is verify that the remote site sent us the right password, all we know is that particular message is authentic. We don't want to have to include the password on every message we send, just as we don't want to use PK to encrypt every message we send. We will use both authentication techniques to establish initial authenticity, then use something else to tie that initial authenticity to subsequent interactions. Let's move right along to SSL/TLS to talk about how we do that, so we don't need to keep promising you that we'll get to it.

## SSL/TLS

We saw in an earlier chapter that a standard method of communicating between processes in modern systems is the socket. That's equally true when the processes are on different machines. So a natural way to add cryptographic protection to communications crossing unprotected networks is to add cryptographic features to sockets. That's precisely what SSL (the Secure Socket Layer) was designed to do, many years ago. Unfortunately, SSL did not get it quite right. That's because it's pretty damned hard to get it right, not because the people who designed and built it were careless. They learned from their mistakes and created a new version of encrypted sockets called Transport Layer Security (TLS). You will frequently hear people talk about using SSL. They are usually treating it as a shorthand for SSL/TLS. SSL, formally, is insecure and should never be used for anything. Use TLS. The only exception is that some very old devices might run software that doesn't support TLS. In that case, it's better to use SSL than nothing. We'll adopt the same shorthand as others from here on, since it's ubiquitous.

The concept behind SSL is simple: move encrypted data through an ordinary socket. You set up a socket, set up a special structure to perform whatever cryptography you want, and hook the output of that structure to the input of the socket. You reverse the process on the other end. What's simple in concept is rather laborious in execution, with a number of steps required to achieve the desired result. There are further complications due to the general nature of SSL. The technology is designed to support a variety of cryptographic operations and many different ciphers, as well as multiple methods to perform key exchange and authentication between the sender and receiver.

The process of adding SSL to your program is intricate, requiring the use of particular libraries and a sequence of calls into those libraries to set up a correct SSL connection. We will not go through those operations step by step here, but you will need to learn about them to make proper use of SSL. Their purpose is, for the most part, to allow a wide range of generality both in the cryptographic options SSL supports and the ways you use those options in your program. For example, these setup calls would allow you to create one set of SSL connections using AES and another using Triple DES, if that's what you needed to do.

One common requirement for setting up an SSL connection that we will go through in a bit more detail is how to securely distribute whatever cryptographic key you will use for the connection you are setting up. Best cryptographic practice calls for you to use a brand new key to encrypt the bulk of your data for each connection you set up. You will use public/private keys for authentication many times, but as we discussed earlier, you need to use symmetric cryptography to encrypt the data once you have authenticated your partner, and you want a fresh key for that. Even if you are running multiple simultaneous SSL connections with the same partner, you want a different symmetric key for each connection.

So what do you need to do to set up a new SSL connection? We won't go through all of the gory details, but, in essence, SSL needs to bootstrap a secure connection based (usually) on symmetric cryptography when no usable symmetric key exists. (You'll hear "usually" and "normally" and "by default" a lot in SSL discussions, because of SSL's

ability to support a very wide range of options, most of which are ordinarily not what you want to do.) The very first step is to start a negotiation between the client and the server. Each party might only be able to handle particular ciphers, secure hashes, key distribution strategies, or authentication schemes, based on what version of SSL they have installed, how it's configured, and how the programs that set up the SSL connection on each side were written. In the most common cases, the negotiation will end in both sides finding some acceptable set of ciphers and techniques that hit a balance between security and performance. For example, they might use RSA with 2048 bit keys for asymmetric cryptography, some form of a Diffie-Hellman key exchange mechanism (see the Aside on this mechanism) to establish a new symmetric key, SHA-1 to generate secure hashes for integrity, and AES with 256 bit keys for bulk encryption. A modern installation of SSL might support 50 or more different combinations of these options.

In some cases, it may be important for you to specify which of these many combinations are acceptable for your system, but often most of them will do, in which case you can let SSL figure out which to use in each case without worrying about it yourself. The negotiation will happen invisibly and SSL will get on with its main business: authenticating at least the server (optionally the client), creating and distributing a new symmetric key, and running the communication through the chosen cipher using that key.

We can use Diffie-Hellman key exchange to create the key (and SSL frequently does), but we need to be sure who we are sharing that key with. SSL offers a number of possibilities for doing so, which include skipping authentication and hoping for the best (not generally a good option, but still supported as of TLS version 1.2). The most common method is for the client to obtain a certificate containing the server's public key and to use the public key in that certificate to verify the authenticity of the server's messages, typically by having the server send it to the client. It is possible for the client to obtain the certificate through some other means, though less common. Note that having the server send the certificate is every bit as secure (or insecure) as having the client obtain the certificate through other means. Certificate security is not based on the method used to transport it, but on the cryptography embedded in the certificate.

With the certificate in hand (however the client got it), the Diffie-Hellman key exchange can now proceed in an authenticated fashion. The server will sign its Diffie-Hellman messages with its private key, which will allow the client to determine that its partner in this key exchange is the correct server. Typically, the client does not provide (or even have) its own certificate, so it cannot sign its Diffie-Hellman messages. This implies that when SSL's Diffie-Hellman key exchange completes, typically the client is pretty sure who the server is, but the server has no clue about the client's identity. (Again, this need not be the case for all uses of SSL. SSL includes connection creation options where both parties know each other's public key and the key exchange is authenticated on both sides. Those options are simply not the most commonly used ones, and particularly are not the ones typically used to secure web browsing.)

## ASIDE: DIFFIE-HELLMAN KEY EXCHANGE

What if you want to share a secret key between two parties, but they can only communicate over an insecure channel, where eavesdroppers can hear anything they say? You might think this is an impossible problem to solve, but you're wrong. Two extremely smart cryptographers named Diffie and Hellman solved this problem years ago, and their solution is in common use. It's called Diffie-Hellman key exchange.

Here's how it works. Let's say Alice and Bob want to share a secret key, but currently don't share anything, other than the ability to send each other messages. First, they agree on two numbers, $n$ (a large prime number) and $g$ (which is primitive *mod* $n$). They can use the insecure channel to do this, since $n$ and $g$ don't need to be secret. Alice chooses a large random integer, say $x$, calculates $X = g^x \bmod n$, and sends $X$ to Bob. Bob independently chooses a large random integer, say $y$, calculates $Y = g^y \bmod n$, and sends $Y$ to Alice. The eavesdroppers can hear $X$ and $Y$, but since Alice and Bob didn't send $x$ or $y$, the eavesdroppers don't know those values.

Alice now computes $k = Y^x \bmod n$, and Bob computes $k = X^y \bmod n$. Alice and Bob get the same value $k$ from these computations. Why? Well, $Y^x \bmod n = (g^y \bmod n)^x \bmod n$, which in turn equals $g^{yx} \bmod n$. $X^y \bmod n = (g^x \bmod n)^y \bmod n = g^{xy} \bmod n$, which is the same thing Alice got. So $k$ is the same and is known to both Alice and Bob.

What about those eavesdroppers? They know $g$, $n$, $X$, and $Y$, but not $x$ or $y$. If they compute $k' = X^Y \bmod n$, they get $g^x \bmod n^{g^y \bmod n} \bmod n$, which is not equal to the $k$ Alice and Bob calculated. They do have an approach to derive $x$ or $y$, which would give them enough information to obtain $k$, but that approach requires them to computer a discrete logarithm. That's a solvable problem, but computationally infeasible for large numbers. So if the prime $n$ is large (and meets other properties), the eavesdroppers are out of luck.

Neat, no? But there is a fly in the ointment, when one considers using Diffie-Hellman over a network. It ensures that you securely share a key with someone, but gives you no assurance of who you're sharing the key with. Maybe Alice is sharing the key with Bob, as she thinks and hopes, but maybe she's sharing it with Mallory, who posed as Bob and injected his own $Y$. Since we usually care who we're in secure communication with, we typically augment Diffie-Hellman with an authentication mechanism to provide the assurance of our partner's identity.

Recalling our discussion earlier in this chapter, it actually isn't a problem for the server to be unsure about the client's identity at this point, in many cases. As we stated earlier, the client will probably want to use a password to authenticate itself, not a public key extracted from a certificate. As long as the server doesn't permit the client to do anything requiring trust before the server obtains and checks the client's password, the server probably doesn't care who the client is, anyway. Many servers offer some services to anonymous clients (such as providing them with publically available information), so as long as they can get a password from the client before proceeding to more sensitive subjects, there is no security problem. So the server can ask the client for a user ID and

password later, at any point after the SSL connection is established. Since creating the SSL connection sets up a symmetric key, the exchange of ID and password can be protected with that key.

## Other Authentication Approaches

While passwords and public keys are the most common ways to authenticate a remote user or machines, there are other options.

One such option is used all the time. After you have authenticated yourself to a web site by providing a password, as we described above, the web site will continue to assume that the authentication is valid. It won't ask for your password every time you click a link or perform some other interaction with it. If your session is encrypted at this point, it could regard your proper use of the cryptography as a form of authentication; but you might even be able to quit your web browser, start it up again, navigate back to that web site, and still be treated as an authenticated user, without a new request for your password. At that point, you're no longer using the same cryptography you used before, since you would have established a new session and set up a new cryptographic key. How did your partner authenticate that you were the one receiving the new key?

In such cases, the site you are working with has chosen to make a security tradeoff. It verified your identity at some time in the past using your password and then relies on another method to authenticate you in the future. A common method is to use *web cookies*. Web cookies are pieces of data that a web site sends to a client with the intention that the client store that data and send it back again whenever the client next communicates with the server. Web cookies are built into most browsers and are handled invisibly, without any user intervention. With proper use of cryptography, a server that has verified the password of a client can create a web cookie that securely stores the client's identity. When the client communicates with the server again, the web browser automatically includes the cookie in the request, which allows the server to verify the client's identity without asking for his password again.

If you spend a few minutes thinking about this authentication approach, you might come up with some possible security problems associated with it. The people designing this technology have dealt with some of these problems, like preventing an eavesdropper from simply using a cookie he copied as it went across the network. However, there are other security problems (like someone other than the legitimate user using the computer that was running the web browser and storing the cookie) that can't be solved with these kinds of cookies, but could have been solved if you required the user to provide the password every time. When you build your own system, you will need to think about these sorts of security tradeoffs yourself. Is it better to make life simpler for your user by not asking for her password except when absolutely necessary, or is it better to provide your user with improved security by frequently requiring proof of her identity? The point isn't that there is one correct answer to this question, but that you need to think about such questions in the design of your system.

There are other authentication options. One example is a challenge/response protocol. The remote machine sends you a challenge, typically in the form of a number. To

authenticate yourself, you must perform some operation on the challenge that produces a response. This should be an operation that only the authentic party can perform, so it probably relies on the use of a secret that party knows, but no one else does. The secret is applied to the challenge, producing the response, which is sent to the server. The server must be able to verify that the proper response has been provided. A different challenge is sent every time, requiring a different response, so attackers gain no advantage by listening to and copying down old challenges and responses. Thus, the challenges and responses need not be encrypted. Challenge/response systems usually perform some kind of cryptographic operation, perhaps a hashing operation, on the challenge plus the secret to produce the response. Such operations are better performed by machines than people, so either your computer calculates the response for you or you have a special hardware token that takes care of it. Either way, a challenge/response system requires pre-arrangement between the challenging machine and the machine trying to authenticate itself. The hardware token or the data secret must have been set up and distributed before the challenge is issued.

Another authentication option is to use an authentication server. In essence, you talk to a server that you trust and that trusts you. The party you wish to authenticate to must also trust the server. The authentication server vouches for your identity in some secure form, usually involving cryptography. The party who needs to authenticate you is able to check the secure information provided by the authentication server and thus determine that the server verified your identity. Since the party you wish to communicate with trusts the authentication server, it now trusts you are who you claim to be. In a vague sense, certificates and CAs are an offline version of such authentication servers. There are more active online versions which involve network interactions of various sorts between the two machines wishing to communicate and one or more authentication servers. Online versions are more responsive to changes in security conditions than offline versions like CAs. An old certificate that should not be honored is hard to get rid of, but an online authentication server can invalidate authentication for a compromised party instantly and apply the changes immediately. The details of such systems can be quite complex, so we will not discuss them in depth. Kerberos is one example of such an online authentication server [NT95].

## Some Higher Level Tools

In some cases, we can achieve desirable security effects by working at a higher level. HTTPS (the cryptographically protected version of the HTTP protocol) and SSH (a competitor to SSL most often used to set up secure sessions with remote computers) are two good examples.

### HTTPS

HTTP, the protocol that supports the World Wide Web, does not have its own security features. Nowadays, though, much sensitive and valuable information is moved over the web, so sending it all unprotected over the network is clearly a bad idea. Rather than come up with a fresh implementation of security for HTTP, however, HTTPS takes the existing HTTP definition and connects it to SSL/TLS. SSL takes care of establishing a secure connection, including authenticating the web server using the certificate approach

discussed earlier and establishing a new symmetric encryption key known only to the client and server. Once the SSL connection is established, all subsequent interactions between the client and server use the secured connection. To a large extent, HTTPS is simply HTTP passed through an SSL connection.

That does not devalue the importance of HTTPS, however. In fact, it is a useful object lesson. Rather than spend years in development and face the possibility of the same kinds of security flaws that other developers of security protocols inevitably find, HTTPS makes direct use of a high quality transport security tool, thus replacing an insecure transport with a highly secure transport at very little development cost.

HTTPS obviously depends heavily on authentication, since we want to be sure we aren't communicating with malicious web sites. HTTPS uses certificates for that purpose. Since HTTPS is intended primarily for use in web browsers, the certificates in question are gathered and managed by the browser. Modern browsers come configured with the public keys of many certificate signing authorities (CAs, as we mentioned earlier). Certificates for web sites are checked against these signing authorities to determine if the certificate is real or bogus. Remember, however, what a certificate actually tells you, assuming it checks out: that at some moment in time the signing authority thoughts it was a good idea to vouch that a particular public key belongs to a particular party. These is no implication that the party is good or evil, that the matching private key is still secret, or even that the certificate signing authority itself is secure and uncompromised, either when it created the certificate or at the moment you check it. There have been real world problems with web certificates in all these cases. Remember also that HTTPS only vouches for authenticity. An authenticated web site using HTTPS can still launch an attack on your client. An authenticated attack, but that won't be much consolation if it succeeds.

While HTTPS is primarily intended to help secure web browsing, it is sometimes used to secure other kinds of communications. Some developers have leveraged HTTP for purposes rather different than standard web browsing, and, for them, using HTTPS to secure their communications is both natural and cheap. However, you can only use HTTPS to secure your system if you commit to using HTTP as your application protocol, and HTTP was intended primarily to support a human-based activity. HTTP messages, for example, are typically encoded in ASCII and include substantial headers designed to support web browsing needs. You may be able to achieve far greater efficiency of your application by using SSL, rather than HTTPS. Or you can use SSH.

**SSH**

SSH stands for "Secure Shell," which accurately describes the original purpose of the program. SSH is available on Linux and other Unix systems, and to some extent on Windows systems. SSH was envisioned as a secure remote shell, but it has been developed into a more general tool for allowing secure interactions between computers. Most commonly this shell is used for command line interfaces, but SSH can support many other forms of secure remote interactions. For example, it can be used to protect remote X Windows sessions. Generally, TCP ports can be forwarded through SSH, providing a powerful method to protect remote interactions.

SSH addresses many of the same problems seen by SSL, often in similar ways. Remote users must be authenticated, shared encryption keys must be established, integrity must be checked, and so on. SSH typically relies on public key cryptography and certificates to authenticate remote servers. Clients frequently do not have their own certificates and private keys, in which case providing a user ID and password is permitted. SSH supports other options for authentication not based on certificates, such as the use of authentication servers (such as Kerberos) and password-based authentication. Various ciphers (both for authentication and for symmetric encryption) are supported, and some form of negotiation is required between the client and the server to choose a suitable set.

SSH is not built on SSL, but is a separate implementation. As a result, the two approaches each have their own bugs, features, and uses. A security flaw found in SSH will not necessarily have any impact on SSL, and vice versa.

## Summary

Distributed systems are critical to modern computing, but are difficult to secure. The cornerstone of providing distributed system security tends to be ensuring that the insecure network connecting system components does not introduce new security problems. Messages sent between the components are encrypted and authenticated, protecting their privacy and integrity, and offering exclusive access to the distributed service to the intended users. Standard tools like SSL/TLS and public keys distributed through X.509 certificates are used to provide these security services. Passwords are often used to authenticate remote human users.

Symmetric cryptography is used for transport of most data, since it is cheaper than asymmetric cryptography. Often, symmetric keys are not shared by system participants before the communication starts, so the first step in the protocol is typically exchanging a key. As discussed in previous chapters, key secrecy is critical in proper use of cryptography, so care is required in the key distribution process. Diffie-Hellman key exchange is commonly used, but it still requires authentication to ensure that only the intended participants know the key.

As mentioned in earlier chapters, building your own cryptographic solutions is challenging and often leads to security failures. A variety of tools, including SSL/TLS, SSH, and HTTPS, have already tackled many of the challenging problems and made good progress in overcoming them. These tools can be used to build other systems, avoiding many of the pitfalls of building cryptography from scratch. However, proper use of even the best security tools depends on an understanding of the tool's purpose and limitations, so developing deeper knowledge of the way such tools can be integrated into one's system is vital to using them to their best advantage.

Remember that these tools only make limited security guarantees. They do not provide the same assurance that an operating system gets when it performs actions locally on hardware under its direct control. Thus, even when using good authentication and encryption tools properly, a system designer is well advised to think carefully about the implications of performing actions requested by a remote site, or providing sensitive

information to that site.  What happens beyond the boundary of the machine the OS controls is always uncertain and thus risky.

## References

[I12]  "Information technology – Open Systems Interconnection – The Directory: Public-key and Attribute Certificate Frameworks"
ITU-T, 2012
*The ITU-T document describing the format and use of an X.509 certificate.  Not recommended for light bedtime reading, but here's where it's all defined.*

[NT94] "Kerberos: An authentication service for computer networks"
B. Clifford Neuman and Theodore Ts'o
IEEE Communications Magazine, Volume 32, No. 9, 1994
*An early paper on Kerberos by its main developers.  There have been new versions of the system and many enhancements and bug fixes, but this paper is still a good discussion of the intricacies of the system.*

[P16]  The International PGP Home Page
http://www.pgpi.org/, 2016.
*A page that links to lots of useful stuff related to PGP, including downloads of free versions of the software, documentation, and discussion of issues related to it.*