Chapter 48: Distributed Systems

- 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 client/server distributed system
- Crux: how do we build systems that work when certain components fail? -> give the illusion that our program never fails
- Tip: Communication is inherently unreliable
    - to 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
- **Performance:** system performance is often critical, with a network connecting our distributed system together, system designers must often think carefully about how to accomplish given tasks -> reduce number of messages sent/increase bandwidth/low latency
- **Security:** Ensure that third parties cannot monitor or alter an ongoing communication between two others
- **Communication:** How should machines within a distributed system communicate with one another?

48.1: Communication Basics

- Communication is fundamentally unreliable
    - Packets are always regularly lost, corrupted, or do not reach their destination
- Multitudes of causes for packet loss or corruption
    - Some bits get flipped due to electrical or other problems during transmission
    - Sometimes an element in the system gets damaged -> such as a work link or packet router or a remote host, etc
    - Fundamental: due to **lack of buffering within a network switch, router, or endpoint**

```
// 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 many packets arrive in the router at once, it is possible that the memory within the router cannot accommodate all of the packets
    - Only choice the router has at this point is to drop the packets

48.2: Unreliable Communication Layers

- One simple way: “we don’t deal with it”
- sometimes useful to let the application communicate through an unreliable layer -> ex: UDP
    - To use **UDP,** a process on other machines (or on same machines) send UDP datagrams to the original process

```
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 48.2: **A Simple UDP Library**

-
    - If you used UDP, you will encounter situations where packets get lost/dropped and thus do not reach their destination -> sender is never warned about the loss
    - This does not mean that UDP does not have guard against any failures at all
        - Some UPD use a basic checksum to detect some forms of packet corruption
    - Using checksums in networking: Before sending a message from one machine to another, compute a checksum over the bytes of the message
        - At the destination, the receiver gets both the checksum and the message and computes a checksum to see whether there was any data corruption

48.3: Reliable Communication Layers

- How does the sender know that the receiver has actually received the message?
- **Acknowledgment/Ack (for short):** The receiver sends a short message back to the sender to acknowledge that the message has been received
- What should the sender do if it does not receive an acknowledgement?

- Need additional mechanism called **timeout**
    - When the sender sends a message, the sender sets a timer to go off after some period of time
    - **If no acknowledgement is received within timeframe, retry**
    - Sender must keep a copy of the message around, in case it needs to send again
- What if the acknowledgement runs into a packet loss?
    - From the receiver side, this means that it could mean that the message is received twice -> means that important packets/data could be overwritten
    - Thus we want to guarantee that each message is received **exactly once by the receiver**
    - To enable the sender to identify each message in a unique way, the receiver needs some way to track whether it has already seen each message before
    - The sender could generate a unique ID for each message, and the receiver can just track every ID its ever seen -> costly + unbounded memory
- **Sequence counter solution:** sender and receiver agree upon a start value
    - whenever a message is sent, the current value of the counter is sent along with the message -> after the message is sent -> increment value by one
    - If the ID of the message matches the counter , then it acks the message and passes it to the application -> receiver increments its own counter
- Most commonly used reliable communication layer is known as **TCP/IP**

48.4: Communication Abstractions

- Tip: be careful setting the timeout value
    - 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 to wait just long enough to detect packet loss but no longer
- What method of communication should we use to build a distributed system?
    - **Distributed Shared Memory:** systems enable processes on different machines to share a large, virtual address space
        - looks like a multithreaded application except that the threads run in different machines
        - When a page is accessed on one machine, two things can happen
            - page is local (on the same machine) and thus can be fetched very quickly
            - page is on some other machine -> page fault occurs -> fetch the page -> install it in the page table of the requesting process -> continue execution
        - Problems with DSM
            - how it handles failure -> if a machine fails, what happens to the pages on that machine?

- What if the data structures of the distributed computation are spread?
- Dealing with failure when parts of the address space go missing is difficult
- Bad performance due to expensive page faults from not being located in the same machine

48.5: Remote Procedure Call (RPC)

- RPC packages want to make the process of executing code on a remote machine as simple and straightforward as calling a local function
    - From the client side, a procedure call is made and the results are simply returned
- **Stub Generator**
    - the stub generator's job is to remove some of the pain of packing function arguments and results into messages by automating it
    - Input into a stub compiler is simply the set of calls a server wishes to export to clients
    - **Generates a client sub that contains each of the functions specified in the interface**
    - Client program wishing to use this RPC would then link this client stub to their application
        - In the code, it simply looks like a function call
    - Steps that occur when making a remote procedure call
        - **Create a message buffer:** a contiguous array of bytes of some size
        - **Pack the needed info into the message buffer:** info includes some kind of identifier for the function to be called -> process of putting all of this info into a single contiguous buffer is called **marshaling** 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
        - **Wait for the reply:** Because function calls are usually **synchronous** we can wait for completion
        - **Unpack return code and other arguments:** also called unmarshaling/deserialization of the message
        - **Return to the caller:** finally just return the client stub back to the client code
    - Code is also generated for the server
        - **Unpack the message:** unmarshaling/deserialization -> takes the information out of the message -> function identifier and argument are extracted
        - **Call into the actual function:** Remote function is actually executed through the RPC runtime calls into the function specified by the ID

        - **Package the results: The return arguments are marshalled backinto a single reply buffer**
        - **Send the reply:** the reply is finally sent to the caller
    - Issues: Organization of the server with regards to concurrency -> may be inefficient if one RPC call blocks server resources are wasted
        - Uses a **thread pool** to have a finite set of threads when the server starts
            - when a message arrives, it is dispatched to one of these worker threads, which then does work of the RPC call, eventually replying
            - During this time, a main thread keeps receiving requests -> maybe dispatches to other workers as well
- **Run-Time Library**
    - Run-time library handles much of the heavy lifting in an RPC system -> most performance and reliability issues are handled here
    - Challenges: how to locate a remote service
        - simplest of approaches build on existing running systems -> hostnames and port numbers provided by internet protocols
    - Once a client knows which server it should talk to for a particular remote service, it must figure out which transport-level protocol that the RPC should be built upon
        - use a reliable port or an unreliable -> TCP/UDP
    - Building RPC on top of a reliable communication layer can lead to a major inefficiency in performance
        - many RPC are actually built on top of unreliable communication layers such as UDP
        - Efficiency increased but the RPC is now responsible for adding extra relibaility
- **Other issues**
    - Some lower-level network protocols provide such sender-side fragmentation (of large packets into a set of smaller ones) and receiver-side reassembly -> if not, the RPC run-time may have to implment functionality to acknowledge requests
    - **Byte ordering:** big endian vs little endian => big endian stores most significant to least significant bits
    - RPC packages handle this by providing a well-defined endianness within their message formats
        - conversion is necessary if endians of message and machine do not match
    - Whether to expose asynchronous nature of communication to clients, thus enabling some performance optimizations
        - Because waits for messages can be very long, some RPC packages enable you to invoke an RPC asynchronously
        - When issued, the RPC package sends the request and returns immediately -> client is free to do other work, such as calls to other RPCs or other useful computation

- The client at some point will want to see the results of the async RPC and will call back into the RPC layer

Distributed Systems Security

Introduction

- An OS can only control its own machine's resource -> thus distributed systems need more security for each respective machine
- Two problems
    - Other machines in the distributed system might not properly implement the security policies you want
    - Machines in a distributed system communicate across a network that none of them can fully trust and therefore cannot control
- How do we protect distributed systems operation?

The Role of Authentication

- We generally can't agree on a particularly policy for all local and remote machines to follow
- We need to authenticate based on a bundle of bits
    - 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 the machine
- Passwords tend to be useful if there are a **vast number of parties** who need to authenticate themselves to **one party**
- Pubic keys tend to be useful if there's **one party** who needs to authenticate himself to a **vast number of parties**
- When a website authenticates itself to a user, it's done with PK cryptography
    - By distributing one single public key to a vast number of users, the website can be authenticated by all its users
    - When a user signs into a website, it is done through a password

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
- Anyone who gets a copy of the bits has the key, but we don't know whose key it is
    - What if we have a trusted other party known to everyone who needs to authenticate our partner -> send a signature along with the cryptographically hashed bunch of bits
- We have a third party create a signed bundle of bits called a **certificate**
    - Essentially, it contains information about the party that owns the public key, the public key itself, and other information such as the expiration date
    - Entire set of information is run through the cryptographic hash, and the result is encrypted with the trusted third party's private key, digitally signing the certificate
- The signing authority did not need to participate in the process of customer checking the certificate

    - Only needs to be done once per customer. After obtaining the certificate and checking it, the customer has the public key he needs -> he can simply store and use it after that point
    - If the customer loses the public key, he can use the certificate to extract it once again
- Chicken and egg problem
    - We'll learn a company's public key by getting a certificate for it
    - The certificate will be signed by a third party such as Verisign or Symantec
    - We'll check the signature by knowing the third party's public key
    - However, where did we get the third party's public key?
        - Ultimately, we have to bootstrap it by obtaining a public key for somebody we trust
    - Where do we get this public key?
        - Most commonly, it comes from pieces of software we obtain and install
        - Browser: comes with public keys for several hundred trust authorities
- If you are building your own distributed system, you can create your own certificates from a machine you trust and can handle the bootstrapping issue by carefully hand-installing the certificate signing machine's public key wherever it needs to be
- ***Remember, the only point of PK certificate is to distribute the public key, so if the public keys are already where they need to be, you don't need certificates

Password Authentication for Distributed Systems

- Works best when there are numerous parties attempting to authenticate into a single party
- Makes sense, for example, when an individual user is authenticating himself into a website that hosts many users such as Amazon
- How do we properly handle password authentication over the network, when it is a reasonable choice?
    - password is usually associated with a particular user ID, so the user provides that ID and password to the site requiring authentication
    - Because network communications cannot guarantee confidentiality, we need to send both the ID and the password to the website

SSL/TLS

- Standard method for communicating between processes in modern systems is the socket
    - we can add cryptographic features to sockets
    - People created SSL (secure socket layer) to provide cryptographic protection in sockets -> is insecure and should never be used for anything
    - However, because SSL did not get it quite right, we also use TLS (transport security layer)
- Concept behind SSL: move encrypted data through an ordinary socket

    - set up a socket, set up a special structure to perform whatever hashing algorithm you want, and hook the output of that structure to the input of the socket
    - Process of setting up SSL is rather complicated: requires use of particular libraries and a sequence of calls into those libraries to set up a correct SSL connection
        - purpose is to allow a wide range of generality in both cryptographic options SSL supports and the ways in which you use those options in your program
    - One common requirement in setting up SSl is to securely distribute the cryptographic key we will use for the connection we are setting up
    - SSL needs to bootstrap a secure connection based on symmetric cryptography when no usable symmetric key exists
        - 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 the SSL connection they have set up
        - The negotiation will set up a set of ciphers, etc. that both parties can agree to use (SHA-3, Diffie-Hellman key exchange, etc.)
    - SSL frequently uses Diffie-Hellman key exchange to create the key, but we need to be sure who we're sharing this key with
        - Most common way is for a 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 message
        - Having the server send the certificate is every bit as secure as having the client obtain the certificate through other means
    - The server will sign its Diffie-Hellman messages with its private key, which allow the client to determine that its partner in this key exchange is the correct server
    - 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
- **Aside: Diffie-Hellman Key Exchange:** Used for when you want to share a secret key between two parties, but they can only communicate over an insecure channel
- **Final Word about SSL/TLS:** a protocol, not a software package
    - If you see a security problem involving SSL, determine whether it is a protocol flaw or an implementation flaw before taking further action -> if implementation flaw we could just use another implementation

Other Authentication Approaches

- When you use a website, it sometimes won't ask you for a password everytime you click a link or perform an 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 -> you might even be able to quit your web browser and start it up again and still be considered an authentication

- However, at this point, we're not using the same cryptographic key -> how are we able to authenticate that you were the one receiving the new key?
    - Site we are working with has chosen to make a security tradeoff -> verified your identity at some time in the past using your password and then relies on another method to authenticate you in the future -> i.e. **web cookies**
    - **Web cookies:** pieces of data that a website 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
- **Challenge/Response protocol:** machine sends you a challenge, typically in the form of a number -> you must perform some operation on the challenge that produces a response -> must be an operation that only the authentic party can perform, so it relies on the use of a secret that party knows but no one else does
- **Using an authentication server:** You talk to a server that you trust and trusts you
    - 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

Some Higher level tools (HTTPS, SSH)

- Protocol that supports the world wide web, does not have its own security features
- 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
- Once the SSL is established, all subsequent interactions between the client and server use the secured connection -> simply pass HTTP through an SSL connection
- Depends heavily on authentication, since we want to be sure that we are not communicating with malicious websites
    - Modern browsers come configured with the public keys of many certificate signing authorities
    - HTTPS only vouches for authenticity so an authenticated website using HTTPS can still launch an attack on your client
- Sometimes used to secure web browsing + other kinds of communications -> HTTPS messaging (ASCII encoded)
- **SSH**
    - Secure Shell -> describes the original purpose of the program -> available on Linux and other Unix systems
    - SSH addresses many of the same problems seen by SSL
        - remote users must be authenticated, shared encryption keys must be checked on as well
        - SSH typically relies on public key cryptography and certificates to authenticate remote servers

    - SSH provides a good example of a common general kind of network security vulnerability called a man-in-the-middle attack
        - Attack occurs when two parties think they are communicating directly, but actually are communicating through a malicious third party without knowing it

Distributed Systems Goals and Challenges (Kampe)

Goals

- **World wide web:** much of the work we do is done in collaboration with others and so we often need to share work products

Client/Server Usage Model

- 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

- We can achieve reliability and availability by combining multiple ordinary already existing devices, like RAID, instead of building new devices out of the best components
- **The key is to distribute service over multiple independent servers**: the key is to only have a single point of failure that would not cause other systems to crash

Scalability

- We need to design systems that can be expanded incrementally, by adding additional computers and storage as they were needed. If our growth plan in to scale-out (rather than to scale-up), we are going to be building our system out of multiple independent computers

Flexibility

- If the components of our service interact with one another through network protocols, it will likely be very easy to change the deployment model -> provides flexibility

Challenges: Why are Distributed Systems Hard to Build

- 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

- 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

- In a single system, it may be easy to tell that one of our systems died, but in a distributed system the only evidence we may have is that we stop receiving messages from it -> much more difficult to diagnose why a system failed
- If we expect a distributed system to continue running when a node has failed, all of the components need to be made more robust. -> components holding resource locks can be difficult to fix

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 timeline (total ordering)
    - at any moment, say what the state of every resource in the system is
- Neither of these are true in a distributed system
    - unless operations are performed by message exchanges, it is generally not possible to say whether a particular operation on mode A happened after a different operation on node B
    - Because of the independence of parallel events, different nodes may at any given instant, consider a single resource in different states -> resource is not a single state but rather a vector of the state that the resource is considered to be in

Complexity of Management

- Although a single computer system has just a single configuration, a thousand different systems may each be configured differently
    - different databases of known users
    - may be configured with different options
    - may have different lists of which servers can perform which functions
    - switches may be configured with different routing and firewall rules
- Even if we configure a distributed management service to push management updates out to all nodes
    - some nodes may not be up when the updates are sent -> does not learn
    - 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 uncover weaknesses that had never caused problems under lighter ones
- Bottlenecks: more nodes mean more messages, which may result in increased overhead, and longer delays -> performance drops/race conditions, etc.

Heterogeneity

- In a single computer system, all applications
    - are running on the same ISA

    - are running the same version of the OS
    - using the same versions of the same libraries
    - directly interact with one-another through the OS
- In a distributed system, each node may
    - run a different ISA
    - run a different OS
    - run different versions of software and protocols

Peter Deutsch's "Seven Fallacies" of Distributed Computing

- **Assumptions that proved to be disastrously false**
- **1) The network is reliable**
    - Subroutine calls always happen and messages/responses aren't guaranteed to be delivered
- **2) Latency is zero**
    - The time to make a subroutine call is negligible -> time for a message exchange could easily be 1000x greater
- **3) Bandwidth is infinite**
    - In-memory data copies can be performed at phenomenal rates. Network throughput is limited, and large numbers of clients can easily saturate NICs, switches, and WAN links
- **4) The network is secure**
    - While not perfect, OS are sufficiently well protected that we seldom have to worry about malicious attacks within our own computer. Once we put our computer on a network it becomes susceptible to penetration attempts, man-in-the-middle attacks, etc.
- **5) Topology doesn't change**
    - Routes change and new clients/servers appear and disappear continuously
- **6) There is one administrator**
    - There may not be a single database of all known clients -> different systems may be administered with different privileges
- **7) Transport Cost is zero**
    - Network infrastructure is not free, and the capital and operational costs of equipment and channels to transport all of our data can dramatically increase the cost of a proposed service
- **8) Network is homogenous (Added later)**
    - nodes on the network are running different versions, of different OS, on machines with different ISA, word lengths, byte orders, etc.

Representational State Transfer (REST)

- **An architectural style that defines a set of constraints and properties based on HTTP**
- Provide interoperability between computer systems on the Internet

- REST compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using an **uniform and predefined set of stateless operations**