Introduction

Distributed systems transformed the world. Connecting to a web server on another planet appears to be a simple version of a client/server distributed system. Behind the scenes, these complicated services are built from a big collection (i.e., thousands) of machines, each of which cooperate to deliver the site's specific function. So now you know what makes researching distributed systems fun. It is worthy of a whole class; we will only introduce a few main points.

Building a distributed system introduces new obstacles. That being said, we do not know how to develop "perfect" components or systems, therefore we focus on failure. How can we make a modern web service appear to clients as though it never fails?

BUILDING SYSTEMS THAT WORK WHEN PARTS FAIL

How can we make a system out of elements that don't always work?
The underlying question is similar to RAID storage arrays, but the difficulties and solutions are more sophisticated.

While failure is a major obstacle in building distributed systems, it also gives an opportunity. Yes, machines fail, but it doesn't mean the whole system fails. We can create a system that looks to work well despite the fact that its components fail frequently. This is the beauty and benefit of distributed systems, and why Google, Facebook, and other current web services rely on them.

Other significant difficulties exist. Given that our distributed system is connected by a network, system designers must often consider ways to limit the quantity of messages delivered and make communication as efficient as possible (low latency, high bandwidth).

Finally, security is a must. A remote site connection requires some confidence that the remote party is who they claim to be. A third party cannot monitor or change an ongoing communication between two others.

A distributed system introduces a new element: communication.
Specifically, how should machines in a distributed system communicate?
We'll start with the most fundamental primitives, messages, and build up from there. As stated above, how should communication layers manage failures?

Communication Basics

OPEN CLIENT.C, SERVER.C

Modern networking assumes that communication is fundamentally unstable. Whether over the Internet or a local high-speed network like Infiniband, packets are frequently lost, corrupted, or otherwise not delivered.

Packet loss or corruption can occur for several reasons. Some bits become inverted during transmission owing to electrical or other issues. A network link, packet router, or even the remote host can be damaged or otherwise not perform properly; network cables can be accidently severed.

However, absence of buffering within a network switch, router, or endpoint causes packet loss. Even if all links operated properly and all system components (switches, routers, end hosts) were up and running as planned, loss is still conceivable for the following reasons. Assume a packet arrives at a router; it must be stored in memory for processing. If a large number of similar packets arrive at the same time, the router's memory may be full. At that time, the router can only drop one or more packets. The same thing happens at the end hosts; sending too many messages to a single machine simply overwhelms its resources, causing packet loss.

Packet loss is fundamental to networking. How should we deal with it?

Unreliable Communication Layers

OPEN UDP.C

Simply put, we don’t deal with it. Because some programs can handle packet loss, it is occasionally desirable to let them connect with a rudimentary unreliable messaging layer (see the Aside at end of chapter). The UDP/IP networking stack found on practically all modern systems is a good example of such an unstable layer. In order to communicate with UDP, a process uses the sockets API to build a communication endpoint (a datagram is a fixed-sized message up to some max size).

Figures 48.1 and 48.2 demonstrate a simple UDP/IP client and server. An email client can send a message to a server, which replies. This little bit of code is all you need to start developing distributed networks!

Unreliable communication layers include UDP. If you use it, you will run into circumstances when packets are lost (dropped) and never reach their destination. That doesn’t mean UDP is immune to failures. So UDP includes a checksum to detect packet tampering.

We need more since many apps want to deliver data without worrying about packet loss. We require dependable communication over an unreliable network.

Reliable Communication Layers

The handling of packet loss requires new mechanisms and strategies. Consider a basic case where a client sends a message to a server across an unreliable link. The first question is: how does the sender know the receiver got the message?

INSERT IMAGE

This method is called an acknowledgment, or ack for short. The sender transmits a message to the receiver, who replies with a short message. As seen in Figure 48.3,
After receiving an acknowledgment, the sender knows that the communication was received. But what if the sender doesn't get an acknowledgement?

INSERT IMAGE

A timeout is required in this scenario. When sending a message, the sender now sets a timer. If no acknowledgement is received within that period, the sender assumes the message is lost. The sender merely resends the identical message, hoping that this time it would be received. To make this work, the sender must store a copy of the message in case it is needed again. Some call the approach timeout/retry because of the timeout and retry; brilliant networking types, eh? Figure 48.4 illustrates.

INSERT IMAGE

Sadly, timeout/retry in this form is insufficient. Figure 48.5 demonstrates a potential problem with packet loss. In this case, the acknowledgement is lost, not the original communication. So, no ack received, timeout and retry are in order. But from the receiver's point of view, the same message has been received twice! While this may be OK in some instances, imagine what happens when you download a file and extra packets are repeated throughout the download. So, while looking for a reliable message layer, we normally want to ensure that each message is only received once.

The sender must uniquely identify each communication, and the recipient must track whether it has already seen each message. Receiver just acknowledges a duplicate transmission, but does not pass it on to the program receiving the data. So the sender gets the ack, but the message doesn't get received twice, retaining the exactly-once semantics.

So many techniques to find duplicate messages. Senders may produce unique IDs for each message, and receivers could monitor every ID seen. However, it is prohibitively expensive, requiring limitless memory to track

all IDs.

A sequence counter is a method that solves this problem quickly and efficiently. With a sequence counter, each side agrees on a start value (e.g., 1) for a counter. Whenever a message is sent, the counter's current value (N) is transmitted along with it as an ID. Sender increases value (to N + 1) after message sent.
The receiver uses its counter as the expected value for the sender's ID. If the message's ID (N) matches the receiver's counter (also N), it accepts the message and forwards it to the application. Recipient updates counter (to N + 1) and awaits next message.

If the sender loses the ack, they timeout and resend N. The receiver's number is now higher (N + 1), indicating it has already received the message. So it acknowledges the message but does not forward it. Sequence counters can be used to avoid duplicates in this way.

TCP/IP, or TCP for short, is the most widely used dependable communication layer. TCP is much more sophisticated than we described above, with mechanisms to handle network congestion, multiple open requests, and hundreds of other minor adjustments and optimizations. If you're curious, take a networking course and master the content.

Communication Abstractions

The next topic in this chapter is: what communication abstraction should we use when creating a distributed system?

Over time, the systems community evolved many techniques. Some work expanded OS concepts to function in a distributed context. DSM systems, for example, allow processes on various computers to share a huge virtual address space. These threads run on multiple machines instead of various processors within the same machine.

Most DSM systems work via the OS's virtual memory mechanism. It can happen on one machine when a page is accessed. In the first (best) instance, the page is already local, therefore the data is promptly fetched. In the second example, the page is on another computer. To continue, the page fault handler contacts another system to fetch the page and install it in the requesting process's page table.

This method is not extensively used today. DSM's biggest flaw is its failure handling. Consider a machine that breaks; what happens to the pages on that system? What if the distributed computation data structures span the full address space? In this circumstance, some data structures would be unavailable. Imagine a linked list where a "next" pointer points into a section of the address space that is missing. Yikes!

Another issue is performance. When writing code, one expects memory access is cheap. Some DSM accesses are cheap, but others generate page errors and expensive remote fetches. Thus, DSM programmers had to carefully design computations so that almost no communication occurred, undermining the purpose of such an approach. Despite much research, DSM is no longer used to develop dependable distributed systems.

Remote Procedure Call (RPC)

PL abstractions make far more sense than OS abstractions for constructing distributed systems. The most common abstraction is a remote procedure call, or RPC.

The purpose of all remote procedure call packages is to make contacting a distant function as straightforward as calling a local function. So a procedure call is made to a client, and the results are returned later. The server merely specifies which routines to export. The RPC system handles the rest of the magic with a stub generator (sometimes called a protocol compiler) and a run-time library. We'll now examine each of these points in more depth.

Stub Genertator

The stub generator's job is to automate the process of packing function parameters and results into messages. Thus, a stub compiler can perhaps optimize such code and hence increase performance.

The compiler receives just the calls a server wishes to export to clients. It might be as simple as this:

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

The stub generator uses an interface like this to generate code. To use this RPC service, a client program must link with this client stub and call into it to make RPCs.

Internally, the client stub functions conduct the entire remote procedure call. This is because the client sees the code as a function call (e.g., func1(x)); internally, the client stub for func1() accomplishes this:

Set up a message buffer. Usually, a message buffer is just an array of bytes.

Pack the necessary information into the message buffer. It includes the identifier of the function to be called and all the arguments it requires (for example, in our example above, one integer for func1). When all of this information is gathered into a single contiguous buffer, the process is called marshaling or serializing.
• Send the message to the destination RPC server. Communication with the RPC server and all the details necessary for it to run correctly is handled by the RPC run-time library, described further below.
• Wait for the response. Since function calls are usually synchronous, the call will wait for its completion.
• Unpack return codes and arguments. This process is straightforward if the function just returns a single return code; however, more complex functions might return more complex results (e.g., a list), so the stub should unpack those as well. The process is also referred to as deserialization or unmarshaling.
• Return to the caller. Lastly, return from the client stub to the client code.

Also, server code is created. The server steps are as follows:

• Unpack the message. Unmarshaling or deserialization removes the information from the incoming message. Arguments and function identifiers are extracted from the message.

• Call into the function. We have reached the point where the remote function is executed. Runtime RPC calls the function specified by the ID and passes the arguments.
• Package your results. The return argument(s) are marshaled back into a single reply buffer.
• Send the reply. The reply is finally sent to the caller.

A stub compiler must also consider a few more factors. To begin, how does one package and convey a complex data structure? An integer file descriptor, a buffer pointer, and a size argument are passed to the write() system call. If an RPC package receives a pointer, it must interpret it and take the appropriate action. Usually, this is done by annotating the data structures with more information, allowing the compiler to know which bytes need to be serialized.

Another critical issue is the server's concurrency organization. A simple server simply waits for requests in a loop, one by one. Likely inefficient because if one RPC call blocks (e.g., on I/O), server resources are squandered. So most servers are built concurrently. A thread pool is a common setup. When the server starts, a finite set of worker threads is created; when a message arrives, it is dispatched to one of these worker threads, which then performs the RPC call, eventually replying; while this is happening, a main thread continues to receive requests, possibly dispatching them to other workers. The standard expenses arise as well, largely in programming complexity, as the RPC calls may now require locks and other synchronization primitives to assure proper operation.

Run-Time Library

The run-time library solves most performance and reliability issues in an RPC system. On to the major issues of constructing such a run-time layer.

Finding a remote service is one of our initial hurdles. This name issue is frequent in distributed systems and is outside the scope of our current discussion. The simplest approaches rely on existing naming systems, such as hostnames and port numbers. So, the client has to know the RPC server's hostname or IP address and the port number it uses (a port number is just a way of identifying a particular communication activity taking place on a machine, allowing multiple communication channels at once). After that, any machine in the system can send packets to a certain address.

Once a client knows which server to contact for a certain service, the question becomes which transport-level protocol to use. Should the RPC system employ a dependable protocol like TCP/IP or an unreliable communication layer like UDP/IP?
The decision appears simple: we want a reliable way to send a request to a remote server and obtain a reliable response. So we should use a reliable protocol like TCP, right?

Sadly, putting RPC on top of a solid communication layer might lead to performance issues. Remember how reliable communication layers work: acknowledgements plus timeout/retry. When a client delivers an RPC request to a server, the server acknowledges the request to let the caller know it was received. Similarly, when the server replies to the client, the client acknowledges receipt. Building an RPC protocol on top of a reliable communication layer sends two "extra" messages.

As a result, many RPC packages rely on unreliable communication protocols like UDP. This allows for a more efficient RPC layer, but also adds the duty of providing reliability. Using timeout/retry and acknowledgments, the RPC layer achieves the necessary amount of responsibility. The communication layer can ensure that each RPC occurs exactly once (no failures) or at most once by employing sequence numbering (in the case where failure arises).

Summary

We have seen the introduction of a new topic, distributed systems, and its fundamental issue: handling failure. Failure is unusual on a desktop machine, but common in a data center with thousands of machines. Managing failure is critical in distributed systems.

A distributed system's heart is communication. The RPC package handles all the nasty details, including timeout/retry and acknowledgment, to produce a service that closely resembles a local procedure call.

The best approach to learn an RPC package is to use it. RPC systems from Sun and Google are both older; Google's gRPC and Apache Thrift are newer. Why not give one a whirl?