CHAPTER 4

Distributed Systems Fundamentals

Scaling a system naturally involves adding multiple independently moving parts. We run our software components on multiple machines and our databases across multiple storage nodes, all in the quest of adding more capacity. Consequently, our solutions are distributed across multiple machines in multiple locations, with each machine processing events concurrently, and exchanging messages over a network.

This fundamental nature of distributed systems has some profound implications on the way we design, build and operate our solutions. This chapter provides the basic 'nature of the beast' information you need to know to appreciate the issues and complexities of distributed software systems. We briefly cover communications networks hardware and software, how to deal with the implications of communications failures, distributed coordination, and the complex issue of time in distributed systems.

Communications Basics

Every distributed system has software components that communicate over a network. If a mobile banking app requests the user's current bank account balance, a (simplified) sequence of communications occurs along the lines of:

1. The cell phone app sends a request over the cellular network addressed to the bank to retrieve the user's bank balance
2. The request is routed across the internet to where the bank's web servers are located.
3. The bank's web server authenticates the request (checks it's really the supposed user) and sends a request to a database server for the account balance.
4. The database server reads the account balance from disk and returns it to the web server
5. The web server sends the balance in a reply message addressed to the app, which is routed over the internet and the cellular network until the balance magically appears on the screen of the mobile device

It almost sounds simple when you read the above, but in reality, there's a huge amount of

complexity hidden beneath this sequence of communications. Let's examine some of these in the following sections.

Communications Hardware

The bank balance request will inevitably traverse multiple different networking technologies and devices. The global internet is a heterogeneous machine, comprising different types of network communications channels and devices that shuttle many millions of messages a second across networks to their intended destinations.

Different types of communications channels exist. The most obvious categorization is wired versus wireless. For each category there are multiple network transmission hardware technologies that can ship bits from one machine to another. Each technology has different characteristics, and the ones we typically care about are speed and range.

For physically wired networks, the two most common types are local area networks (LANs) and wide area networks (WANs). LANs are networks that can connect devices at 'building scale', being able to transmit data over a small number (e.g. 1-2) of kilometers. Contemporary LANs can transport between 100 megabits per second (Mbps) to 1 gigabits per second (Gbps). This is known as the network's bandwidth, or capacity. The time taken to transmit a message across a LAN – the network's latency – is sub-millisecond with modern LAN technologies.

WANs are networks that traverse the globe and make up what we collectively call the internet. These long-distance connections are the high speed data 'pipelines' connecting cites and countries and continents with fiber optic cables. These cables support a networking technology known as wavelength division multiplexing[1] which makes it possible to transmit up 171 Gbps over 400 different channels, giving more than 70 Terabits per second (Tbps) of total bandwidth for a single fiber link. The fiber cables that span the world normally comprise four or more strands of fiber, giving bandwidth capacity of hundreds of Tbps for each cable.

Latency is more complicated with WANs however. WANs transmit data over 100s to 1000s of kilometers, and the maximum speed that the data can travel is the theoretical speed of light. In reality, fiber optic cables can't reach the speed of light, but do get pretty close to it as we can see in Table 1.

Path	Distance	Time - Speed of Light	Time - Fiber Optic Cable
New York to San Francisco	4,148 km	14 ms	21 ms
New York to London	5,585 km	19 ms	28 ms
New York to Sydney	15,993 km	53 ms	80 ms

Table 1 WAN Speeds

Actual times will be slower than this as the data needs to pass through networking equipment known as routers[2]. Routers are responsible for transmitting data on the physical network connections to ensure data is transmitted across the internet from source to destination. Routers are specialized, high speed devices that can handle several hundred Gbps of network traffic,

[1] https://en.wikipedia.org/wiki/Wavelength-division_multiplexing#:~:text=In%20fiber%2Doptic%20communications%2C%20wavelength,%2C%20colors)%20of%20laser%20light.

[2] https://en.wikipedia.org/wiki/Core_router

pulling data off incoming connections and sending the data out to different outgoing network connections based on their destination. Routers at the core of the internet comprise racks of these devices and hence can process 10s to hundreds of Tbps. This is how you and 1000's of your friends get to watch a steady video stream on Netflix.

Wireless technologies have different range and bandwidth characteristics. Wi-Fi routers that we are all familiar with in our homes and offices are wireless ethernet networks and use 802.11 protocols to send and receive data. The most widely used Wi-Fi protocol, 802.11ac, allows for maximum (theoretical) data rates of up to 5,400Mbps. The most recent 802.11ax protocol, also known as Wi-Fi 6, is an evolution of 802.11ac technology that promises increased throughput speeds of up to 9.6Gbps. The range of Wi-Fi routers is of the order of 10's of meters, and of course is affected by physical impediments like walls and floors.

Cellular wireless technology uses radio waves to send data from our phones to routers mounted on cell towers, which are generally connected by wires to the core internet for message routing. Each cellular technology introduces improved bandwidth and other dimensions of performance. The most common technology at the time of writing is 4G LTE wireless broadband. 4G LTE is around 10 times faster than the older 3G, able to handle sustained download speeds around 10 Mbps (peak download speeds are nearer 50 Mbps) and upload speeds between 2 and 5 Mbps.

Emerging 5G cellular networks promise 10x bandwidth improvements over existing 4G, with 1-2 millisecond latencies between devices and cell towers. This is a great improvement over 4G latencies which are in the 20-40 millisecond range. The trade-off is range. 5G base station range operates at about 500m maximum, whereas 4G provides reliable reception at distances of 10-15kms.

This whole collection of different hardware types for networking comes together in the global internet. The internet is heterogeneous network, with many different operators globally and every type of hardware imaginable. Figure 1 (from Wikipedia[3]) shows a simplified view of the major components that comprise the internet. Tier 1 networks are the global high-speed internet backbone. There are around 20 Tier 1 Internet Service Providers (ISPs) who manage and control global traffic. Tier 2 ISPs are typically regional (e.g. one country), have lower bandwidth than Tier 1 ISPs, and deliver content to customers through Tier 3 ISPs. Tier 3 ISPs are the ones that charge your exorbitant fees for your home internet every month.

[3] https://en.wikipedia.org/wiki/Tier_1_network#List_of_Tier_1_networks

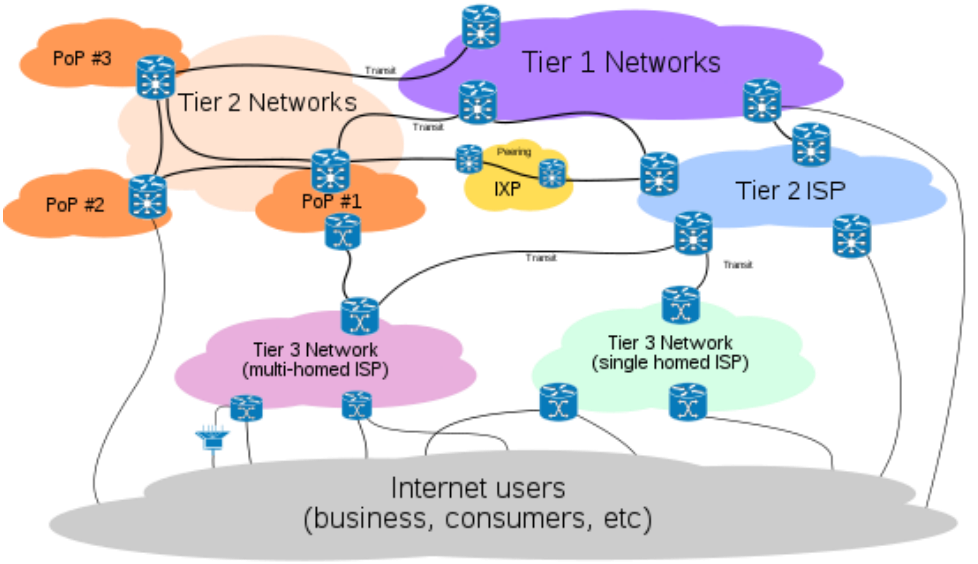

Figure 1 Simplified view of the Internet (from Wikipedia)

There's a lot more complexity to how the internet works than described here. That complexity is beyond the scope of this chapter. From a distributed systems software perspective, we need to understand more about the 'magic' that enables all this hardware to route messages from say my cell phone, to my bank and back. This is where the IP protocol comes in.

Communications Software

Software systems on the internet communicate using the Internet Protocol suite. The Internet Protocol suite specifies host addressing, data transmission formats, message routing and delivery characteristics. There are four abstract layers, which contain related protocols that support the functionality required at that layer. These are, from lowest to highest:

1) the data link layer, specifying communication methods for data across a single network segment. This is implemented by the device drivers and network cards that live inside your devices.
2) the internet layer specifies addressing and routing protocols that make it possible for traffic to traverse the independently managed and controlled networks that comprise the internet. This is the IP protocol in the internet protocol suite.
3) the transport layer, specifying protocols for reliable and best-effort host-to-host communications. This is where the well-known TCP and UDP protocols live.
4) the application layer, which comprises several application level protocols such as HTTP and SCP.

Each of the higher layer protocols builds on the features of the lower layers. The enables the IP suite to support end to end data communications across the internet. In the following, we'll briefly cover the IP protocol for host discovery and message routing, and the TCP and UDP transport protocols that can be utilized by distributed applications.

Internet Protocol (IP)

IP defines how hosts are assigned addresses on the internet and how messages are transmitted

between two hosts who know each other's addresses.

Every device on the internet has its own address. These are known as Internet Protocol (IP) addresses. The location of an IP address can be found using an internet wide directory service known as Domain Naming Service (DNS). DNS is a widely distributed, hierarchical database that acts as the address book of the internet.

The technology currently used to assign IP addresses, known as Internet Protocol version 4 (IPv4), will eventually be replaced by its successor, IPv6. IPv4 is a 32-bit addressing scheme that before long will run out of addresses due to the number of devices connecting to the internet. IPv6 is a 128-bit scheme that will offer an (almost) infinite number of IP addresses. As an indicator, in July 2020 about 33% of the traffic processed by Google.com[4] is IPv6.

DNS servers are organized hierarchically. A small number of root DNS servers, which are highly replicated, are the starting point for resolving and an IP address. When an internet browser tries to find a web site, a network host known as the local DNS server that is managed by your employer or ISP, will contact a root server with the requested host name. The root server replies with a referral to a so-called *authoritative* DNS server that manages name resolution for, in our banking example, *.com* addresses. There is an authoritative name server for each top-level internet domain (e.g. .com, .org, .net, etc).

Next the local DNS server will query the *.com* DNS server, which will reply with the address of the DNS server which knows about all the IP addresses managed by mybank.com. This DNS is queried, and it returns the actual IP address we need to communicate with the application. The overall scheme is illustrated in Figure 2.

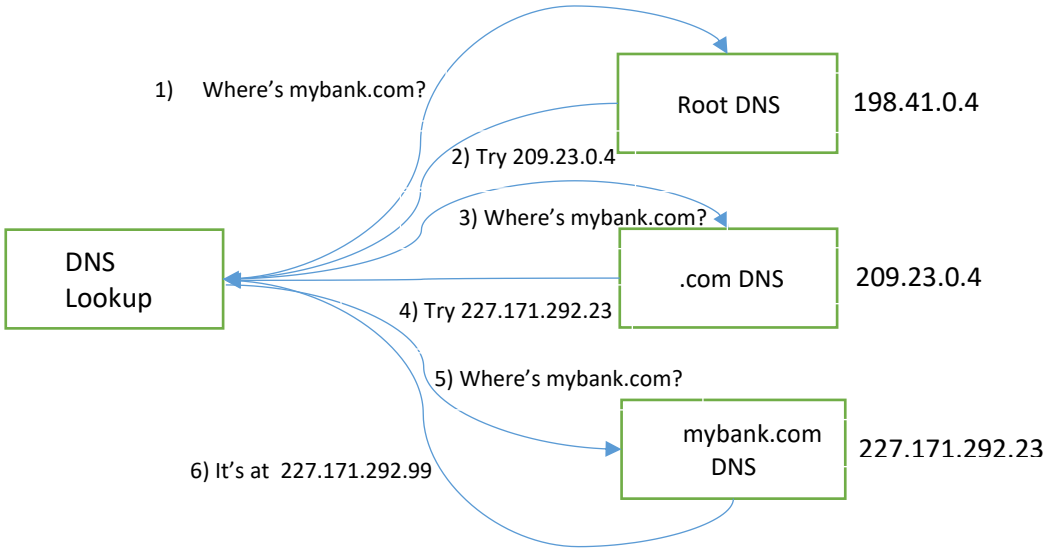

Figure 2 Example DNS Lookup for mybank.com

The whole DNS database is highly geographically replicated so there are no single points of failure, and requests are spread across multiple physical servers. Local DNS servers also remember the IP addresses of recently contacted hosts, which is possible as IP addresses don't change very often. This means the complete name resolution process doesn't occur for every site we contact.

[4] https://www.google.com/intl/en/ipv6/statistics.html

Armed with a destination IP address, a host can start sending data across the network as a series of IP packets. IP has the task of delivering data from the source to the destination host based on the IP addresses in the packet headers. IP defines a packet structure that contains the data to be delivered, along with header data including source and destination IP addresses. Data sent by an application is broken up into a series of packets which are independently transmitted across the Internet.

IP is known as a best-effort delivery protocol. This means it does not attempt to compensate for the various error conditions that can occur during packet transmission. Possible transmission errors include data corruption, packet loss and duplication. In addition, as every packet is routed from source to destination independently, different packets may be delivered to the same destination via different network paths, resulting in out-of-order delivery to the receiver. Treating every packet independently is known as packet-switching. This allows the network to dynamically respond to conditions such as link failure and congestion, and hence is a defining characteristic of the internet.

Because of this design, the IP is unreliable. If two hosts require reliable data transmission, they need to add additional features to make this occur. This is where the next layer in the IP protocol suite, the transport layer, enters the scene.

Transmission Control Protocol (TCP)

Once an application or browser has discovered the IP address of the server it wishes to communicate with, it can send messages using the transport protocol API. This is achieved using Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), which are the established standard transport protocols for the IP network stack.

Distributed applications can choose which of these protocols to use. APIs for both TCP and UDP are widely available in mainstream programming languages such as Java, Python and C++. In reality, use of these APIs is not common as higher-level programming abstractions hide the details from most applications. In fact, the IP protocol suite application layer contains several of these application level APIs, including HTTP, which is very widely used in mainstream distributed systems. Still, it's important to understand TCP, UDP and their differences.

Most requests on the internet are sent using TCP. TCP is:

- connection-oriented
- stream-oriented
- reliable

TCP is a connection-oriented protocol. Before any messages are exchanged between applications, TCP uses a 3-step handshake to establish a two-way connection between the client and server applications. The connection stays open until the TCP client calls close to terminate the connection with the TCP server. The server responds by acknowledging the close request before the connection is dropped.

Once a connection is established, a client sends a sequence of requests to the server as a data stream. When a data stream is sent over TCP, it is broken up into individual network packets, with a maximum packet size of 65535 bytes. Each packet contains a source and destination address, which is used by the underlying IP protocol to route the messages across the network.

The internet is a packet-switched network, which means every packet is individually routed across the network. The route each packet traverses can vary dynamically based on the conditions in the network, such as link congestion or failure. This means the packets may not arrive at the server in the same order they are sent from the client. To solve this problem, a TCP sender

includes a sequence number in each packet so the receiver can reassemble packets into a stream that is identical to the order they were sent.

Reliability is needed as network packets can be lost or delayed during transmission between sender and receiver. To achieve reliable packet delivery, TCP uses a cumulative acknowledgement mechanism. This means a receiver will periodically send an acknowledgement packet that contains the highest sequence number of the received packets. This implicitly acknowledges all packets sent with a lower sequence number, meaning all have been successfully received. If a sender doesn't receive an acknowledgement within a timeout period, the packet is resent.

TCP has many other features, such as checksums to check packet integrity, and dynamic flow control to ensure a sender doesn't overwhelm a slow receiver by sending data too quickly. Along with connection establishment and acknowledgments, this makes TCP a relatively heavyweight protocol, which trades off reliable over efficiency.

This is where UDP comes into the picture. UDP is a simple connectionless protocol, which exposes the user's program to any unreliability of the underlying network. There is no guarantee of in order delivery, or even delivery for that matter. It can be thought of as a thin veneer (layer) on top of the underlying IP protocol, and deliberately trades off raw performance over reliability. This however is highly appropriate for many modern applications where the odd lost packet has very little effect. Think streaming movies, video conferencing and gaming, where one lost packet is unlikely to be perceptible by a user.

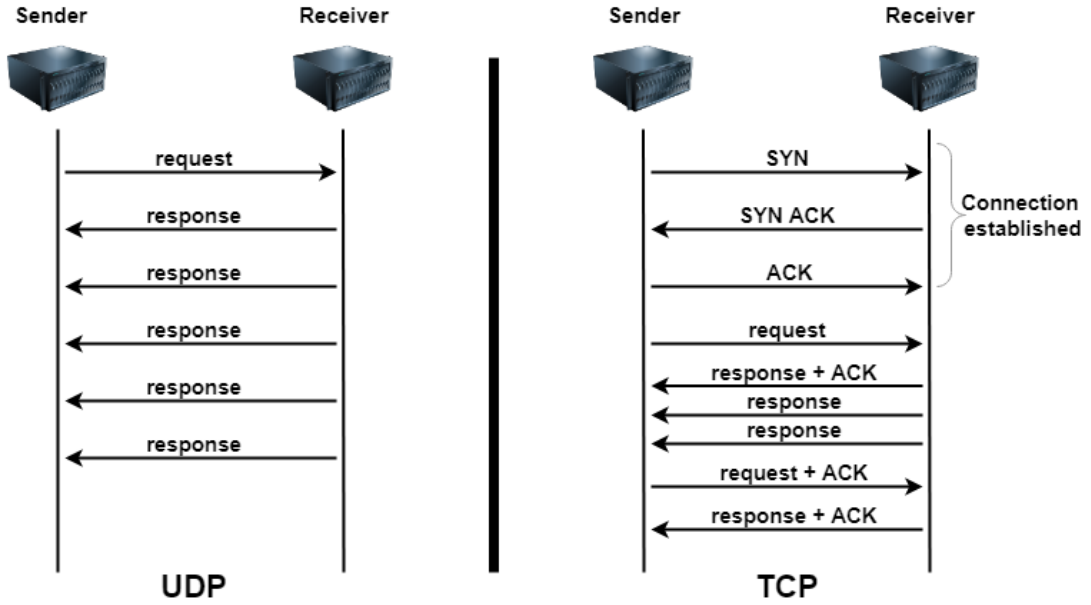

Figure 3 Comparing TCP and UDP

Figure 3 depicts some of the major differences between TCP and UDP. TCP incorporates a connection establishment 3-packet handshake, and piggybacks acknowledgements (ACK) of packets so that any packet loss can be handled by the protocol. There's also a TCP connection close phase involving a 4-way handshake that is not shown in the diagram. UDP dispenses with connection establishment, tear down, acknowledgements and retries. Hence applications using UDP need to be tolerant of packet loss and client or server failures and behave accordingly.

Remote Method Invocation

It's perfectly feasible to write our distributed applications using APIs that interact directly with transport layer protocols like TCP. If we use for example the TCP sockets library, we can create a connection, known as a socket, between a client and a server and exchange data over that connection.

A socket is basically a 'pipe' between the client and server. Once the socket is created, the client sends data to the server in a stream. In our bank example, the client might request a balance for the user's checking account. Ignoring specific language issues (and security!!), the client might send a message payload as follows over a socket to the server:

{"balance", "000169990"}

In this message, "balance" represents the operation we want the server to execute, and "000169990" is the bank account number.

In the server, we need to know that the first string in the message is the operation identifier, and based on this value being "balance", the second is the bank account number. The server then uses these values to presumably query a database, retrieve the balance and send back the results, perhaps as a message formatted with the account number and current balance, as below:

{"000169990", "220.77"}

In any complex system, the server will support many operations. In mybank.com, we might have for example "login", "transfer", "address", "statement", "transactions", and so on. Each will be followed by different message payloads that the server needs to interpret correctly to fulfill the client's request.

What we are defining here is an *application specific protocol.* As long as we send the necessary values in the correct order for each operation, the server will be able to respond correctly. If we have an erroneous client that doesn't adhere to our application protocol, well, our server needs to do thorough error checking.

Stepping back, if we were defining the mybank.com server in an object-oriented language such as Java, we would have each operation it can process as a method, which is passed an appropriate parameter list for that operation, as shown in Figure 4.

```
1.  // Simple mybank.com server interface
2.  public interface MyBank {
3.      public float balance  (String accNo);
4.      public boolean  statement(String month) ;
5.      // other operations
6.   }
```

Figure 4 Simple mybank.com server interface

There are several advantages of having such an interface, namely:

- Calls from the client to the server can be statically checked by the compiler to ensure they are of the correct type
- Changes in the server interface (e.g. add a new parameter) force changes in the client code to adhere to the new method signature
- The interface is clearly defined by the class definition, and hence straightforward for a

client programmer to utilize

These benefits of an explicit interface are of course well known in sequential programming. The whole discipline of object-oriented design is pretty much based upon these foundations, where an interface defines a contract between the caller and callee. Compared to the implicit, application protocol we need to program to with sockets, the advantages are significant.

This fact was recognized reasonably early in the evolution of distributed systems. Since the early 1990's, we have seen an evolution of technologies that enable us to define explicit server interfaces and call these across the network using essentially the same syntax as we would in a sequential program. A summary of the major approaches is given in Table 2. Collectively they are known as Remote Procedure Call (RPC) or Remote Method Invocation (RMI) technologies.

Technology	Dates	Main features
Distributed Computing Environment (DCE)[5]	Early 1990s	DCE RPC provides a standardized approach for client-server systems. Primary languages were C/C++.
Common Object Request Broker Architecture (CORBA)[6]	Early 1990s	Facilitates language-neutral client-server communications based on an object-oriented Interface Definition Language (IDL). Primary language support in C/C++, Java, Python, Ada.
Java Remote Method Invocation (RMI)[7]	Late 1990s	A pure Java-based remote method invocation that facilitates distributed client-server systems with the same semantics as Java objects.
XML Web Services	2000	Supports client-server communications based on HTTP and XML. Servers define their remote interface in the Web Services Description Language (WSDL)

Table 2 Summary of major RPC/RMI Technologies

While the syntax and semantics of these RPC/RMI technologies vary, the essence of how each operates is the same. Let's continue with our Java example of mybank.com to use this as an example of the whole classification.

Using Java RMI, we can trivially make our `MyBank` interface in Figure 4 a remote interface, as shown in Figure 5.

```
1.  import java.rmi.*;
2.  // Simple mybank.com server interface
3.  public interface MyBank extends Remote{
4.      public float balance  (String accNo)
5.           throws RemoteException;
6.      public boolean  statement(String month)
7.           throws RemoteException ;
8.      // other operations
9.   }
```

[5] http://www.opengroup.org/dce/

[6] http://www.corba.org

[7] https://docs.oracle.com/javase/9/docs/specs/rmi/index.html

Figure 5 Java RMI Example

The empty `java.rmi.Remote` interface serves as a marker to inform the Java compiler we are creating an RMI server. In addition, each method must throw `java.rmi.RemoteException`. These exceptions represent errors that can occur when a distributed call between two objects is invoked over a network. The most common reasons for such an exception would be a communications failure or the server object having crashed.

We then must provide a class that implements this remote interface. Figure 6 shows an extract of the server implementation - the complete code for this example is in the Chapter 4 code repository. Points to note are:

- The server extends the `UnicastRemoteObject` class. This essentially provides the functionality to instantiate remotely callable object.
- Once the server object is constructed, its availability must be advertised to remote clients. This is achieved by storing a reference to the object in the *RMI Registry,* and associating a logical name with it – in this example, *MyBankServer*. The registry is a simple directory service that enables clients to look up the location (network address and object reference) of and obtain a reference to an RMI server.

```
1.  public class MyBankServer extends UnicastRemoteObject
2.                              implements MyBank  {
3.              // constructor/method implementations omitted
4.     public static void main(String args[]){
5.          try{
6.            MyBankServer server=new MyBankServer();
7.            // create a registry in local JVM on default port
8.            Registry registry = LocateRegistry.createRegistry(1099);
9.            registry.bind("MyBankServer", server);
10.           System.out.println("server ready");
11.         }catch(Exception e){
12.                 // code omitted for brevity}
13.         }
14.    }
```

Figure 6 Implementing an RMI Server

An extract from the client code to connect to the server is shown in Figure 7. It obtains a reference to the remote object by performing a `lookup` operation (line 3) in the RMI Registry and specifying the logical name that identifies the server. The reference returned by the lookup operation can then be used to call the server object in the same manner a local object. However there is a difference – the client must be ready to catch a `RemoteException` that will be thrown by the Java runtime when the server object cannot be reached.

```
1.   // obtain a remote reference to the server
2.   MyBank bankServer=
3.           (MyBank)Naming.lookup("rmi://localhost:1099/MyBankServer");
4.   //now we can call the server
5.   System.out.println(bankServer.balance("00169990"));
```

Figure 7 Implementing an RMI Client

Figure 8 depicts the call sequence amongst the components that comprise a RMI system. The

Stub and *Skeleton* are compiler-generated objects that facilitate the remote communications. The skeleton is a TCP network endpoint (host, port) that listens for calls to the associated server. The sequence of operations is as follows:

1. When the server reference is stored in the RMI Registry, the entry contains the client stub that can be used to make remote calls to the server.
2. The client queries the registry, and the stub for the server is returned.
3. The client stub accepts a method call to the server interface from the client
4. The stub transforms the request into a network request to the server host. This transformation process is known as marshalling.
5. The skeleton accepts network requests from the client, and unmarshalls the network packet data into a valid call to the server object implementation
6. The skeleton waits for the method to return a result
7. The skeleton marshalls the method results into a network reply packet that is sent the client
8. The stub unmarshalls the data passes the result to the client call site

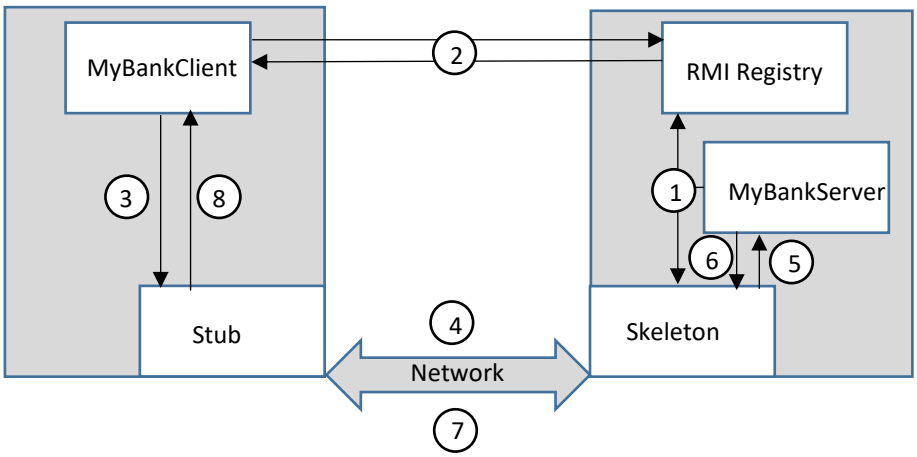

Figure 8 Schematic depicting the call sequence for establishing a connection and making a call to a RMI server object

This Java RMI example illustrates the basics that are used for implementing any RPC/RMI mechanism, even in modern languages like Erlang[8] and Go[9]. Regardless of implementation, the basic attraction of these approaches to is to provide an abstract calling mechanism that provides *location transparency* for clients making remote server calls.

RPC/RMI is not without its flaws. Marshalling and unmarshalling can become inefficient for complex object parameters. Cross language marshalling – client in one language, server in another – can cause problems due to types being represented differently in different languages, causing subtle incompatibilities. And if a remote method signature changes, all clients need to obtain a new compatible stub which can be cumbersome in large deployments.

For these reasons, most modern systems are built around simpler protocols based on HTTP

[8] http://erlang.org/doc/man/rpc.html
[9] https://golang.org/pkg/net/rpc/

and using JSON for parameter representation. Instead of operation names, HTTP verbs (PUT, GET, POST, etc) have associated semantics that are mapped to a specific URL. This approach originated in the work by Roy Fielding on the REST approach[10]. REST has a set of semantics that comprise a RESTful architecture style, and in reality, most systems do not adhere to these. We'll discuss REST and HTTP API mechanisms in the Chapter 5.

Partial Failures

The components of distributed systems communicate over a network. In communications technology terminology, the shared local and wide area networks that our systems communicate over are known as *asynchronous* networks. With asynchronous networks:

- nodes can choose to send data to other nodes at any time
- The network is *half-duplex*, meaning that one node sends a request and must wait for a response from the other. These are two separate communications.
- The time for data to be communicated between nodes is variable, due to reasons like network congestion, dynamic packet routing and transient network connection failures
- The receiving node may not be available due to a software or machine crash
- Data can be lost. In wireless networks, packets can be corrupted and hence dropped due to weak signals or interference. Internet routers can drop packets during congestion[11].
- Nodes do not have identical internal clocks, hence they are not synchronized

What does this mean for our applications? Well, put simply, when a client sends a request to server, how long does it wait until it receives a reply? Is the server node just being slow? Is the network congested and the packet has been dropped? If the client doesn't get a reply, what should it do?

Let's explore these scenarios in detail. The core problem is known as handling partial failures, and the general situation is depicted in Figure 9.

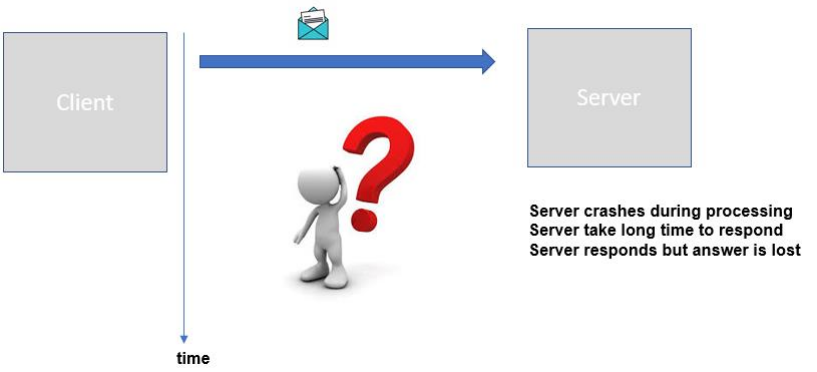

Figure 9 Handling Partial Failures

[10] Fielding, Roy Thomas (2000). "Architectural Styles and the Design of Network-based Software Architectures". Dissertation. University of California, Irvine.

[11] https://en.wikipedia.org/wiki/Packet_loss

When a client node sends a network request to a server node and expects a response, the following outcomes may occur:

1. The request succeeds and a rapid response is received. All is good.
2. The destination IP address lookup may fail. In this case the client rapidly receives a error message.
3. The IP address is valid but the destination node or target server process has failed. Again the sender will receive an error message.
4. The request is received by the target server, which fails while processing the request and no response is ever sent.
5. The request is received by the target server, which is heavily loaded. It processes the request but takes a long time (e.g 24 seconds!) to respond
6. The request is received by the target server and a response is sent. However, the response is not received by the client due to a network failure.

Numbers (1) to (3) are easy for the client to handle, as a response is received rapidly. A result from the server or an error message – either allows the client to proceed. Failures that can be detected quickly are easy to deal with.

Numbers (4) to (6) pose a problem for the client. It has no insight into the reason why a response has not been received. From the client's perspective, these three outcomes look the same. It cannot know, without waiting forever, whether the response will arrive eventually, or never arrive. And waiting forever doesn't get much work done. More insidiously, the client cannot know if the operation succeeded and server or network failure caused the result to never arrive, or if the request is on its way - delayed simply due to congestion in the network/server. These faults are collectively known as *crash faults*[12].

The typical solution that clients adopt to handle crash faults is to resend the request after a configured timeout period. This however is fraught with danger, as Figure 10 illustrates. The client sends a request to the server to deposit money in a bank account. When it receives no response after a timeout period, it resends the request. What is the resulting balance? The server may have applied the deposit, and it may not, depending on the partial failure scenario.

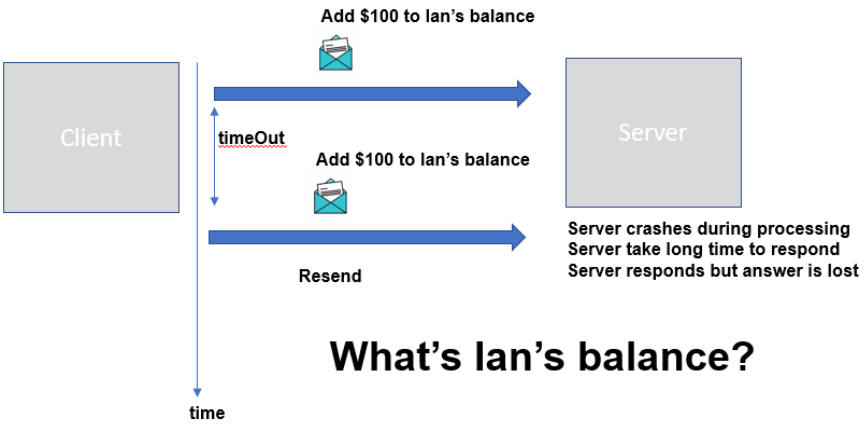

Figure 10 Client retries a request after timeout

The chance that the deposit may occur twice is a fine outcome for the customer. The bank

[12] https://medium.com/baseds/modes-of-failure-part-1-6687504bfed6#

though is unlikely to be amused by this possibility. Hence we need a way to ensure in our server operations that retried, duplicate requests from clients only result in the request being applied once. This is necessary to maintain correct application semantics. This property is known as idempotence. Idempotent operations can be applied multiple times without changing the result beyond the initial application. This means that for the example in Figure 10, the client can retry the request as many times as it likes, and the account will only be increased by $100.

Requests that make no persistent state changes are naturally idempotent. This means all read requests are inherently safe and no extra work is needed in the server. Updates are a different matter. The system needs to devise a mechanism such that duplicate client requests can be detected by the server, and they do not cause any state changes. In API terms, these endpoints cause mutation of the server state and must be idempotent.

The general approach to building idempotent operations is as follows:

- Clients include a unique *idempotence-key* in all requests that mutate state. The key identifies a single operation from the specific client or event source. It is usually a composite of a user identifier, such as the session key, and a unique value such as a local timestamp, UUID or a sequence number.
- When the server receives a request, it checks to see if it has previously seen the idempotence key value by reading from a database that is uniquely designed for implementing idempotence. If the key is not in the database, this is a new request. The server therefore performs the business logic to update the application state. It also stores the idempotence key in a database to indicate that the operation has been successfully applied.
- If the idempotence key is in the database, this indicates that this request is a retry from the client and hence should not be processed. In this case the server returns a valid response for the operation so that (hopefully) the client won't retry again.

The database used to store idempotence keys can be implemented in, for example:

- a separate database table or collection in the transactional database used for the application data
- a dedicated database that provides very low latency lookups, such as a simple key-value store

Unlike application data, idempotence keys don't have to be retained forever. Once a client receives an acknowledgement of a success for an individual operation, the idempotence key can be discarded. The simplest way to achieve this is to automatically remove idempotence keys from the store after a specific time period, such as 60 minutes or 24 hours, depending on application needs and request volumes.

In addition, an idempotent API implementation must ensure that the application state is modified, **and** the idempotence key is stored. Both must occur for success. If the application state is modified and, due to some failure, the idempotent key is not stored, then a retry will cause the operation to be applied twice. If the idempotence key is stored but for some reason the application state is not modified, then the operation has not been applied. If a retry arrives, it will be filtered out as duplicate as the idempotence key already exists, and the update will be lost.

The implication here is that the updates to the application state and idempotence key store must **both** occur, or **neither** must occur. If you know your databases, you'll recognize this as a requirement for transactional semantics. (We'll discuss how distributed transaction are achieved in Chapter XXX) This will ensure *exactly-once semantics*, which guarantees that all messages will always be processed exactly once – what we need for idempotence.

Exactly once does not mean that there are no message transmission failures, retries and no application crashes. These are all inevitable. The important thing is that the retries eventually succeed and the result is the always the same.

We'll return to the issue of communications delivery guarantees in later chapters. As Figure 11 illustrates, there's a spectrum of semantics, each with different guarantees and performance characteristics. *At most once* delivery is fast and unreliable – this is what the UDP protocol provides. *At least once* delivery is the guarantee provided by TCP/IP, meaning duplicates are inevitable. *Exactly-once* delivery, as we've discussed here, requires guarding against duplicates and hence trades off reliability against slower performance.

As we'll see, some advanced communications mechanisms can provide our applications with exactly once semantics. However, these don't operate at Internet scale because of the performance implications. That is why, as our applications are built on the at least once semantics of TCP/IP, we must implement exactly once semantics in our APIs that cause state mutation.

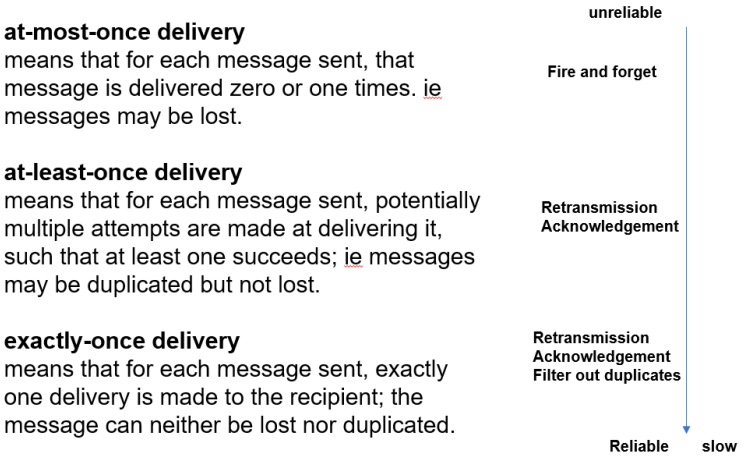

Figure 11 Communications Delivery Guarantees

Consensus in Distributed Systems

Crash faults have another implication for the way we build distributed systems. This is best illustrated by the Two Generals Problem[13], which is illustrated in Figure 12.

Imagine a city under siege by two armies. The armies lie on opposite sides of the city, and the terrain surrounding the city is difficult to travel through and visible to snipers in the city. In order to overwhelm the city, it's crucial that both armies attack at the same time. This will stretch the city's defenses and make victory more likely for the attackers. If only one army attacks, then they will likely be repelled. How can the two generals reach agreement on the exact time to attack, such that both generals know for certain that agreement has been reached? They both need certainty that the other army will attack at the agreed time.

To coordinate an attack, the first general sends a messenger to the other, with instructions to attack at a specific time. As the messenger may be captured or killed by snipers, the sending general cannot be certain the message has arrived unless they get an acknowledgement messenger from the second general. Of course, the acknowledgement messenger may be captured or killed, so even if the original messenger does get through, the first general may never know. And even if

[13] https://en.wikipedia.org/wiki/Two_Generals%27_Problem

the acknowledgement message arrives, how does the second general know this, unless they get an acknowledgement from the first general?

Hopefully the problem is apparent. With messengers being randomly captured or extinguished, there is no guarantee the two generals will ever reach consensus on the attack time. In fact, it can be proven that it is not possible to *guarantee* agreement will be reached. There are solutions that increase the likelihood of reaching consensus. For example, Game of Thrones style, each general may send 100 different messengers every time, thus increasing the probability that at least one will make the perilous journey to the other friendly army and successfully deliver the message.

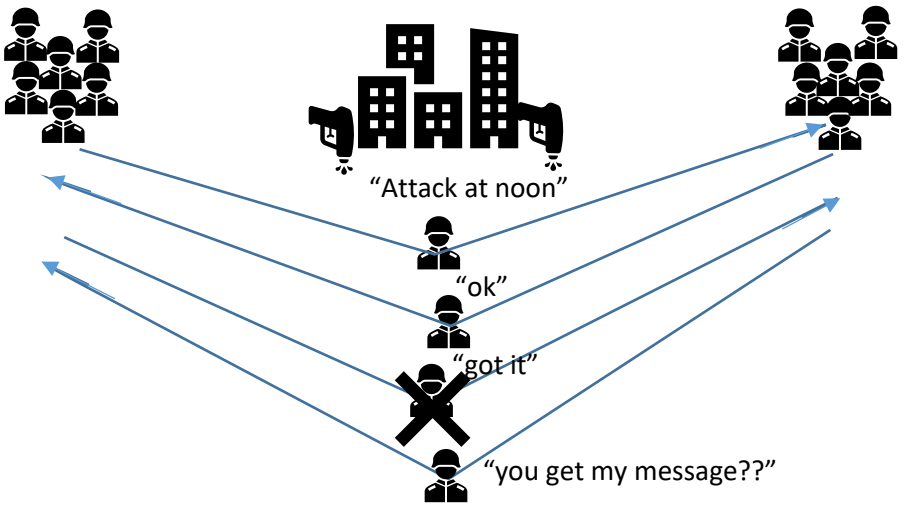

Figure 12 The Two Generals Problem

The Two Generals problem is analogous to two nodes in a distributed system wishing to reach agreement on some state, such as the value of a data item that can be updated at either. Partial failures are analogous to losing messages and acknowledgements. Messages may be lost or delayed for an indeterminate period of time.

In fact it can be demonstrated that consensus on an asynchronous network in the presence of crash faults, where messages can be delayed but not lost, is impossible to achieve within bounded time. This is known as the FLP Impossibility Theorem[14].

Luckily, this is only theoretical limitation, demonstrating it's not possible to *guarantee* consensus will be reached with unbounded message delays on an asynchronous network. In reality, distributed systems reach consensus all the time. This is possible because while our networks are asynchronous, we can establish sensible practical bounds on message delays and retry after a timeout period. FLP is therefore a worst-case scenario, and as we'll discuss algorithms for establishing consensus when we discuss distributed databases.

Finally, we should note the issue of Byzantine failures. Imagine extending the Two Generals problem to N Generals, who need to agree on a time to attack. However, in this scenario, traitorous messengers may change the value of the time of the attack, or a traitorous general may

[14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2 (April 1985), 374–382.

sense false information to other generals. This class of *malicious* failures are known as Byzantine faults and are particularly sinister in distributed systems. Luckily, the systems we discuss in this book typically live behind well-protected, secure enterprise networks and administrative environments. This means we can practically exclude handling Byzantine faults. Algorithms that do address malicious behavior exist, and if you are interested in a practical example, take a look at Blockchain technologies[15] and BitCoin[16].

Time in Distributed Systems

Every node in a distributed system has its own internal clock. If all the clocks on every machine were perfectly synchronized, we could always simply compare the timestamps on events across nodes to determine the precise order they occurred in. If this were reality, many of the problems we'll discuss with distributed systems would pretty much go away.

Unfortunately, this is not the case. Clocks on individual nodes *drift* due to environmental conditions like changes in temperature or voltage. The amount of drift varies on every machine, but values like 10-20 seconds a day are not uncommon. Or with my current coffee machine, about 15 minutes a day!

If left unchecked, clock drift would render the time on a node meaningless – like my coffee machine if I don't correct it every few days. To address this problem, a number of *time services* exist. A time service represents an accurate time source, such as a GPS or atomic clock, which can be used to reset the clock on a node to correct for drift on packet-switched, variable-latency data networks.

The most widely used time service is NTP[17], which provides a hierarchically organized collection of time servers spanning the globe. The root servers, of which there are around 300 worldwide, are the most accurate. Time servers in the next level of the hierarchy (approximately 20,000) synchronize to within a few milliseconds of the root server periodically, and so on throughout the hierarchy, a with a maximum of 15 levels. Globally there are more than 175,000 NTP servers.

Using the NTP protocol, a node in an application running the NTP client can synchronize to a NTP server. The time on a node is set by a UDP message exchange with one or more NTP servers. Messages are timestamped and through the message exchange the time taken for transit is estimated. This becomes a factor in the algorithm used by NTP to establish what the time on the client should be reset to. A simple NTP configuration is shown in Figure 13. On a LAN, machines can synchronize to an NTP server within a small number of milliseconds accuracy.

[15] https://medium.com/@chrshmmmr/consensus-in-blockchain-systems-in-short-691fc7d1fefe
[16] https://ieeexplore.ieee.org/abstract/document/8123011
[17] www.ntp.org

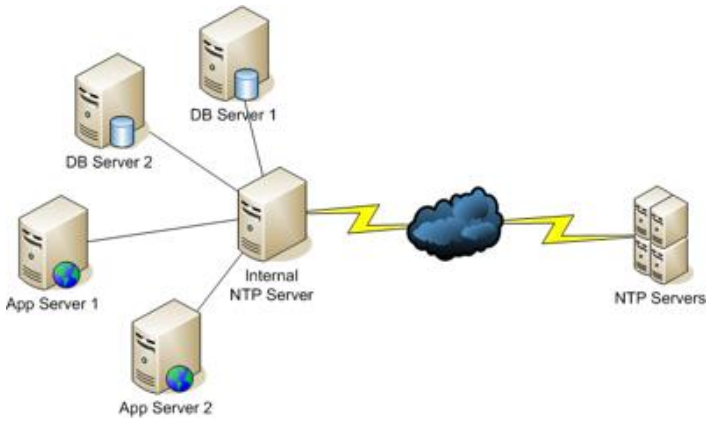

Figure 13 Illustrating using the NTP Service

One interesting effect of NTP synchronization for our applications is that the resetting of the clock can move the local node time forwards or backwards. This means that if our application is measuring the time taken for events to occur (e.g. to calculate event latencies), it is possible that the end time of the event may be earlier than the start time if the `ntp` protocol has set the local time backwards.

In fact, a node has two clocks. These are:

- **Time of Day Clock**: This represents the number of milliseconds since midnight January 1970. In Java, you can get the current time using `System.currentTimeMillis()`. This is the clock that can be reset by NTP, and hence may jump forwards or backwards if it is a long way ahead of NTP time.
- **Monotonic Clock:** This represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past, such as the last time the system was restarted. It will only ever move forward, however it again may not be a totally accurate measure of elapsed time because it stalls during a virtual machine suspension. In Java, you can get the current monotonic clock time using `System.nanoTime()`.

.

Applications can use an NTP service to ensure the clocks on every node in the system are closely synchronized. It's typical for an application to resynchronize clocks on anything from a one hour to one day time interval. This ensures the clocks remain close in value. Still, if an application really needs to precisely know the order of events that occur on different nodes, clock drift is going to make this fraught with danger.

There are other time services that provide higher accuracy than NTP. Chrony[18] supports the NTP protocol but provides much higher accuracy and greater scalability than NTP – the reason it has recently been adopted by Facebook[19]. Amazon has built the Amazon Time Sync Service by installing GPS and atomic clocks in its data centers. This service is available for free to all Amazon cloud customers.

The take away from this discussion is that our applications cannot rely on timestamps of events on different node s to represent the actual order of these events. Clock drift even by a second or two makes cross-node timestamps meaningless to compare. The implications of this will become clear when we start to discuss distributed databases in detail.

[1818] https://chrony.tuxfamily.org/

[19] https://engineering.fb.com/production-engineering/ntp-service/

Summary and Further Reading

This chapter has covered a lot of ground to explain some of the fundamental issues communications and time in distributed systems. The key issues that should resonate with the you, dear reader, are as follows:

1. Communications in distributed systems can transparently traverse many different types of underlying physical networks – e.g. Wi-Fi, wireless, WANs and LANs.
2. Communication latencies are highly variable, and influenced by the physical distance between nodes, and transient network congestion.
3. Communications can fail due to network communications fabric and router failures that make nodes unavailable, and individual node failure.
4. The sockets library has two variants, one that supports TCP/IP for reliable connection-based communications (`SOCK_STREAM`), and one for unreliable UDP datagram-based communications (`SOCK_DGRAM`).
5. RMI/RPC technologies build on TCP/IP sockets to provide abstractions for client-server communications that mirror making local method/procedure calls.
6. Achieving agreement, or consensus on state across multiple nodes in the presence of crash faults is not possible in bounded time on asynchronous networks. Luckily, real networks, especially LANs, are fast and mostly reliable, meaning we can devise algorithms that achieve consensus in practice.
7. There is no reliable global time source that nodes in an application can rely upon to synchronize their behavior. Clocks on individual nodes vary and cannot be used for meaningful comparisons.

These issues will pervade the discussions in the rest of this book. Many of the unique problems and solutions that are adopted in distributed systems stem from these fundamentals. There's no escaping them!

An excellent source for more detailed, more theoretical coverage of all aspects of distributed systems is *George Colouris et at, Distributed Systems: Concepts and Design, 5th Edition, Pearson, 2011.*

Likewise for computer networking, you'll find out all you wanted to know and no doubt more in *James Kurose, Keith Ross, Computer Networking: A Top-Down Approach, 7th Edition, Pearson 2017.*