48

Distributed Systems

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

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

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

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

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

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

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

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

48.1 Communication Basics

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

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

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

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

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

Figure 48.1: **Example UDP Code (`client.c`, `server.c`)**

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

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

48.2 Unreliable Communication Layers

One simple way is this: we don't deal with it. Because some applications know how to deal with packet loss, it is sometimes useful to let them communicate with a basic unreliable messaging layer, an example of the **end-to-end argument** one often hears about (see the **Aside** at end of chapter). One excellent example of such an unreliable layer is found

```
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); // network byte order
  struct in_addr *in_addr;
  struct hostent *host_entry;
  if ((host_entry = gethostbyname(hostname)) == NULL)
    return -1;
  in_addr = (struct in_addr *) host_entry->h_addr;
  addr->sin_addr = *in_addr;
  return 0;
}

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

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 (`udp.c`)**

TIP: USE CHECKSUMS FOR INTEGRITY

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

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

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

in the **UDP/IP** networking stack available today on virtually all modern systems. To use UDP, a process uses the **sockets** API in order to create a **communication endpoint**; processes on other machines (or on the same machine) send UDP **datagrams** to the original process (a datagram is a fixed-sized message up to some max size).

Figures 48.1 and 48.2 show a simple client and server built on top of UDP/IP. The client can send a message to the server, which then responds with a reply. With this small amount of code, you have all you need to begin building distributed systems!

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

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

48.3 Reliable Communication Layers

To build a reliable communication layer, we need some new mechanisms and techniques to handle packet loss. Let us consider a simple

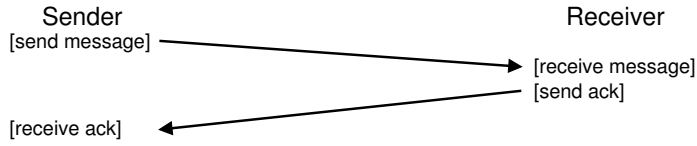

Figure 48.3: **Message Plus Acknowledgment**

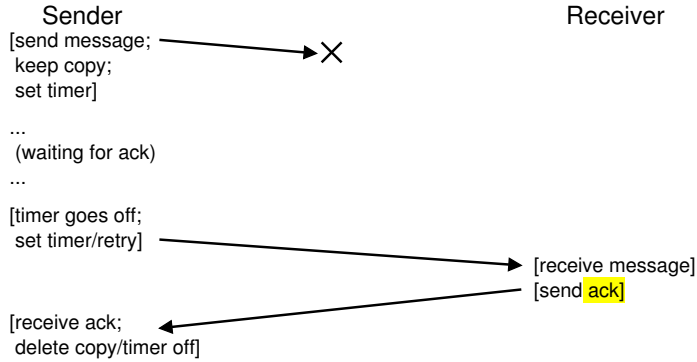

Figure 48.4: **Message Plus Acknowledgment: Dropped Request**

example in which a client is sending a message to a server over an unreliable connection. The first question we must answer: how does the sender know that the receiver has actually received the message?

The technique that we will use is known as an **acknowledgment**, or **ack** for short. The idea is simple: the sender sends a message to the receiver; the receiver then sends a short message back to *acknowledge* its receipt. Figure 48.3 depicts the process.

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

To handle this case, we need an additional mechanism, known as a **timeout**. When the sender sends a message, the sender now sets a timer to go off after some period of time. If, in that time, no acknowledgment has been received, the sender concludes that the message has been lost. The sender then simply performs a **retry** of the send, sending the same message again with hopes that this time, it will get through. For this approach to work, the sender must keep a copy of the message around, in case it needs to send it again. The combination of the timeout and the retry have led some to call the approach **timeout/retry**; pretty clever crowd, those networking types, no? Figure 48.4 shows an example.

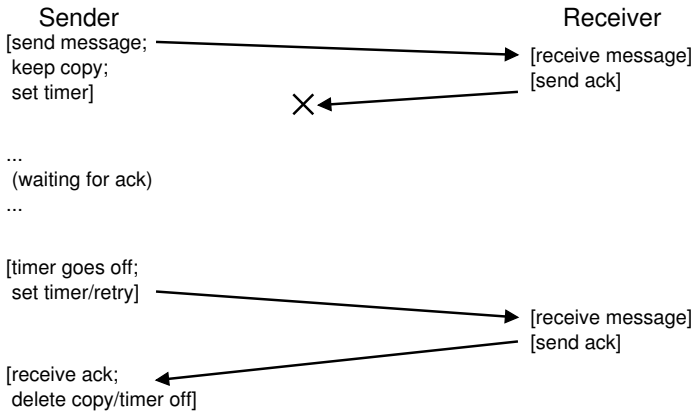

Figure 48.5: **Message Plus Acknowledgment: Dropped Reply**

Unfortunately, timeout/retry in this form is not quite enough. Figure 48.5 shows an example of packet loss which could lead to trouble. In this example, it is not the original message that gets lost, but the acknowledgment. From the perspective of the sender, the situation seems the same: no ack was received, and thus a timeout and retry are in order. But from the perspective of the receiver, it is quite different: now the same message has been received twice! While there may be cases where this is OK, in general it is not; imagine what would happen when you are downloading a file and extra packets are repeated inside the download. Thus, when we are aiming for a reliable message layer, we also usually want to guarantee that each message is received **exactly once** by the receiver.

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

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

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

TIP: BE CAREFUL SETTING THE TIMEOUT VALUE

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

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

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

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

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

48.4 Communication Abstractions

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

The systems community developed a number of approaches over the years. One body of work took OS abstractions and extended them to

operate in a distributed environment. For example, **distributed shared memory** (**DSM**) systems enable processes on different machines to share a large, virtual address space [LH89]. This abstraction turns a distributed computation into something that looks like a multi-threaded application; the only difference is that these threads run on different machines instead of different processors within the same machine.

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

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

A further problem is performance. One usually assumes, when writing code, that access to memory is cheap. In DSM systems, some accesses are inexpensive, but others cause page faults and expensive fetches from remote machines. Thus, programmers of such DSM systems had to be very careful to organize computations such that almost no communication occurred at all, defeating much of the point of such an approach. Though much research was performed in this space, there was little practical impact; nobody builds reliable distributed systems using DSM today.

48.5 Remote Procedure Call (RPC)

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

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

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

Stub Generator

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

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

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

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

Internally, each of these functions in the client stub do all of the work needed to perform the remote procedure call. To the client, the code just appears as a function call (e.g., the client calls `func1(x)`); internally, the code in the client stub for `func1()` does this:

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

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

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

There are a few other important issues to consider in a stub compiler. The first is complex arguments, i.e., how does one package and send a complex data structure? For example, when one calls the `write()` system call, one passes in three arguments: an integer file descriptor, a pointer to a buffer, and a size indicating how many bytes (starting at the pointer) are to be written. If an RPC package is passed a pointer, it needs to be able to figure out how to interpret that pointer, and perform the correct action. Usually this is accomplished through either well-known types (e.g., a `buffer_t` that is used to pass chunks of data given a size, which the RPC compiler understands), or by annotating the data structures with more information, enabling the compiler to know which bytes need to be serialized.

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

Run-Time Library

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

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

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

Naively the choice would seem easy: clearly we would like for a request to be reliably delivered to the remote server, and clearly we would like to reliably receive a reply. Thus we should choose the reliable transport protocol such as TCP, right?

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

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

Other Issues

There are some other issues an RPC run-time must handle as well. For example, what happens when a remote call takes a long time to complete? Given our timeout machinery, a long-running remote call might appear as a failure to a client, thus triggering a retry, and thus the need for some care here. One solution is to use an explicit acknowledgment

Aside: THE END-TO-END ARGUMENT

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

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

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

(from the receiver to sender) when the reply isn't immediately generated; this lets the client know the server received the request. Then, after some time has passed, the client can periodically ask whether the server is still working on the request; if the server keeps saying "yes", the client should be happy and continue to wait (after all, sometimes a procedure call can take a long time to finish executing).

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

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

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

A final issue is whether to expose the asynchronous nature of communication to clients, thus enabling some performance optimizations. Specifically, typical RPCs are made **synchronously**, i.e., when a client issues the procedure call, it must wait for the procedure call to return before continuing. Because this wait can be long, and because the client may have other work it could be doing, some RPC packages enable you to invoke an RPC **asynchronously**. When an asynchronous RPC is issued, the RPC package sends the request and returns immediately; the client is then free to do other work, such as call other RPCs or other useful computation. The client at some point will want to see the results of the asynchronous RPC; it thus calls back into the RPC layer, telling it to wait for outstanding RPCs to complete, at which point return arguments can be accessed.

48.6 Summary

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

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

The best way to really understand an RPC package is of course to use one yourself. Sun's RPC system, using the stub compiler `rpcgen`, is an older one; Google's gRPC and Apache Thrift are modern takes on the same. Try one out, and see what all the fuss is about.

References

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

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

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

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

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

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

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

Homework (Code)

In this section, we'll write some simple communication code to get you familiar with the task of doing so. Have fun!

Questions

1. Using the code provided in the chapter, build a simple UDP-based server and client. The server should receive messages from the client, and reply with an acknowledgment. In this first attempt, do not add any retransmission or robustness (assume that communication works perfectly). Run this on a single machine for testing; later, run it on two different machines.
2. Turn your code into a **communication library**. Specifically, make your own API, with send and receive calls, as well as other API calls as needed. Rewrite your client and server to use your library instead of raw socket calls.
3. Add reliable communication to your burgeoning communication library, in the form of **timeout/retry**. Specifically, your library should make a copy of any message that it is going to send. When sending it, it should start a timer, so it can track how long it has been since the message was sent. On the receiver, the library should **acknowledge** received messages. The client send should **block** when sending, i.e., it should wait until the message has been acknowledged before returning. It should also be willing to retry sending indefinitely. The maximum message size should be that of the largest single message you can send with UDP. Finally, be sure to perform timeout/retry efficiently by putting the caller to sleep until either an ack arrives or the transmission times out; do *not* spin and waste the CPU!
4. Make your library more efficient and feature-filled. First, add very-large message transfer. Specifically, although the network limit maximum message size, your library should take a message of arbitrarily large size and transfer it from client to server. The client should transmit these large messages in pieces to the server; the server-side library code should assemble received fragments into the contiguous whole, and pass the single large buffer to the waiting server code.
5. Do the above again, but with high performance. Instead of sending each fragment one at a time, you should rapidly send many pieces, thus allowing the network to be much more highly utilized. To do so, carefully mark each piece of the transfer so that the re-assembly on the receiver side does not scramble the message.
6. A final implementation challenge: asynchronous message send with in-order delivery. That is, the client should be able to repeatedly call send to send one message after the other; the receiver should call receive and get each message in order, reliably; many messages from

the sender should be able to be in flight concurrently. Also add a sender-side call that enables a client to wait for all outstanding messages to be acknowledged.

7. Now, one more pain point: measurement. Measure the bandwidth of each of your approaches; how much data can you transfer between two different machines, at what rate? Also measure latency: for single packet send and acknowledgment, how quickly does it finish? Finally, do your numbers look reasonable? What did you expect? How can you better set your expectations so as to know if there is a problem, or that your code is working well?